Object-oriented programming with Java

Dr. Constantinos Constantinides

Department of Computer Science and Software Engineering
Concordia University

Classes and objects

- A class is a template from which objects may be created.
  - Can have any number of instances (objects).
- An object contains state (data) and behavior (methods).
- Methods of an object collectively characterize its behavior.
  - Methods can only be invoked by sending messages to an object.
  - Behavior is shared among objects.

2

Identifying objects and their state in a library information system

```
public class Book {

  String author;
  String title;
  String year;

  Book (String author, String title, String year) {
    this.author = author;
    this.title = title;
    this.year = year;
  }

  public void display () {
    System.out.println ("Author: " + author + "\n" +
        "Title: " + title + "\n" +
        "Year: " + year + "\n");
  }

}
```

- author, title, year are instance variables; they hold data.
- They are of type String, i.e. they can hold textual data.
- The state of the object is composed of a set of attributes (or fields), and their current values.

3

Object behavior: methods

```
public class Book {

  String author;
  String title;
  String year;

  Book (String author, String title, String year) {
    this.author = author;
    this.title = title;
    this.year = year;
  }

  public void display () {
    System.out.println ("Author: " + author + "\n" +
        "Title: " + title + "\n" +
        "Year: " + year + "\n");
  }

}
```

- display is of type void, because it does not return any value.
- The body of a method lies between { and } and defines some computation to be done.
- The behavior of the object is defined by a set of methods (functions), which may access or manipulate the state.

4

Object Behavior: constructor methods

```
public class Book {

  String author;
  String title;
  String year;

  Book (String author, String title, String year) {
    this.author = author;
    this.title = title;
    this.year = year;
  }

  public void display () {
    System.out.println ("Author: " + author + "\n" +
        "Title: " + title + "\n" +
        "Year: " + year + "\n");
  }

}
```

- Book is a special method, called the constructor of the class; used to create and initialize instances (objects).
- A constructor is a special method which initializes an object immediately upon creation.
  - It has the exact same name as the class in which it resides.
  - A constructor has no return type, not even void.
- During object creation, the constructor is automatically called to initialize the object.

5

Field initialization during construction

```
public class Book {

  String author;
  String title;
  String year;

  Book (String author, String title, String year) {
    this.author = author;
    this.title = title;
    this.year = year;
  }

…

}
```

- What happens when an object is initialized in Java:
  - All data fields are set to zero, false or null.
  - The data fields with initializers are set, in the order in which they appear in the class definition.
  - The constructor body is executed.

6

Field shadowing

```
public class Book {

  String author;
  String title;
  String year;

  Book (String author, String title, String year) {
    String author = author;
    String title = title;
    String year = year;
  }

…

}
```

- The statement `String author = author;` in the constructor body defines a new local variable author that shadows the data field `author`!
- After the constructor is finished, the local variables are forgotten and the data field `author` is still null (as it was before entering the constructor)

7

Implementing methods

```
public class PurchaseOrder {
   …
   public double calculateTotal (double price,
                                             int quantity) {
   if (quantity >= 0)
      return price * quantity;
   }
}
```

- What is wrong with the following code?
- The path of `quantity < 0` is not terminated by a return statement.
- As a result, a compilation error will occur!

8

Implementing methods (cont.)

```
public class PurchaseOrder {
  // …
  public double calculateTotal (double price,
                                int quantity) {
    double total;
    if (quantity >= 0)
      return unitPrice * quantity;
    return total;
  }
}
```

- What is wrong with the following code?

- Local variables are not automatically initialized to their default values.
- Local variables must be explicitly initialized.
- The code will cause a compilation error.

9

Parameter passing

```
public class IntRef {
  public int val;
  public IntRef(int i) {val = i;}
}
```

```
public class C {
  public void inc(IntRef i) {i.val++;}
}
```

```
C c = new C();
IntRef k = new IntRef(1);        // k.val is 1
c.inc(k);                        // now k.val is 2
```

- <u>All method parameters are passed by value</u> (i.e. modifications to parameters of primitive types are made on copies of the actual parameters).
- <u>Objects are passed by reference</u>.
- In order for a parameter of primitive type to serve as a reference parameter, it must be wrapped inside a class.

10

Parameter passing (cont.)

```
void aMethod(final IntRef i) {
   …
   i = new IntRef(2);        // not allowed
}
```

```
void aMethod(final IntRef i) {
   …
   i.val++;                  // ok
}
```

- A final parameter of a method may not be assigned a new value in the body of the method.
- However, if the parameter is of reference type, it is allowed to modify the object (or array) referenced by the final parameter.

11

Object features

```
public class Book {

  String author;
  String title;
  String year;

  Book (String author, String title, String year) {
    this.author = author;
    this.title = title;
    this.year = year;
  }

  public void display () {
    System.out.println ("Author: " + author + "\n" +
       "Title: " + title + "\n" +
       "Year: " + year + "\n");
  }

}
```

- We distinguish between mutator methods (operations), which change an object, and accessor methods, which merely read its data fields.
  - `display()` is an accessor method.
- The features of an object refer to the combination of the state and the behavior of the object.

12

Type signature

```
public class Book {

  String author;
  String title;
  String year;

  Book (String author, String title, String year) {
    this.author = author;
    this.title = title;
    this.year = year;
  }

  public void display () {
    System.out.println ("Author: " + author + "\n" +
        "Title: " + title + "\n" +
        "Year: " + year + "\n");
  }

}
```

- The <u>type signature of a method</u> (or constructor) is a sequence that consists of the types of its parameters.
  – Note that the return type, parameter names, and final designations of parameters are not part of the signature.
  – Parameter order is significant.

Book - (String, String, String)
display - ()

13

Static features

```
public class staticTest {
        static int a = 3;
        static int b;
        static void method (int x) {
                System.out.println(“x = “+ x);
        }
        static {
                System.out.println("inside static block");
                b = a * 4;
                System.out.println(b);
        }
        public static void main(String[] args) {
                method(42);
        }
}
```

```
inside static block
12
x = 42
```

- Static features are used outside of the context of any instances.
- <u>Static blocks</u>: As soon as the class is loaded, all static blocks are run before `main()`
- Static methods:
  – Static methods can be accessed from any object;
  – They can be called even without a class instantiation, e.g. `main()`
  – Java’s equivalent of global functions.

14

Accessing static features

- Instance variables and methods can be accessed only through an object reference (You cannot access instance variables or call instance methods from static methods!)
- Static fields and methods may be accessed through either an object reference or the class name.

```
objectReference.staticMethod(parameters)
objectReference.staticField

ClassName.staticMethod(Parameters)
ClassName.staticField
```

15

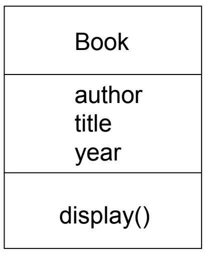
Example on accessing static features

- Each time a Counter object is created, the static variable `howMany` is incremented.
- Unlike the field `value`, which can have a different value for each instance of Counter, the static field `howMany` is universal to the class.

```
public class Counter {
	public Counter() { howMany++; }
	public void reset() { value = 0; }
	public void get() { return value; }
	public void click() { value = (value + 1) % 100; }
	public static int howMany() {return howMany;}
	private int value;
	private static int howMany = 0;
}
```

16

Defining a Book class



```
public class Book {

  String author;
  String title;
  String year;

  Book (String author, String title, String year) {
    this.author = author;
    this.title = title;
    this.year = year;
  }

  public void display () {
    System.out.println ("Author: " + author + "\n" +
        "Title: " + title + "\n" +
        "Year: " + year + "\n");
  }

}
```

Creating a Book instance (object)

```
public class TestV01 {
  static public void main(String args[]) {

    Book MyBook = new Book ("Timothy Budd",
                            "OOP",
                            "1998");
  }
}
```

- The `new` operator creates an instance of a class (object).

Sending messages

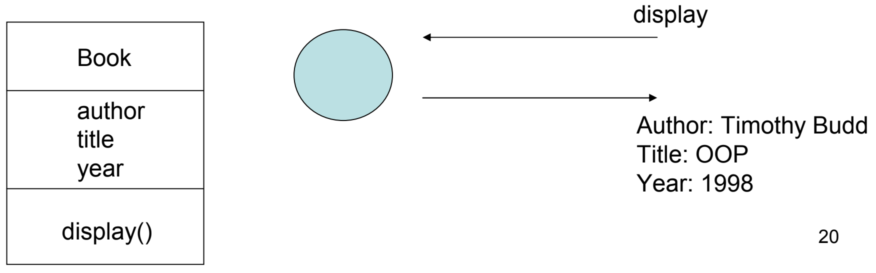
```
public class TestV01 {
  static public void main(String args[]) {

    Book MyBook = new Book ("Timothy Budd",
                            "OOP",
                            "1998");
    MyBook.display();

  }
}
```

- A message represents a command sent to an object (recipient or receiving object, or receiver of the message) to perform an action by invoking one of the methods of the recipient.
- A message consists of the receiving object, the method to be invoked, and (optionally) the arguments to the method.
  - object.method(arguments);

19

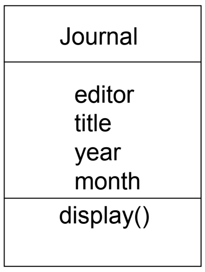
Sending a message to a Book instance

```
public class TestV01 {
  static public void main(String args[]) {

    Book MyBook = new Book ("Timothy Budd", "OOP", "1998");
    MyBook.display();

  }
}
```



20

Defining a Journal class

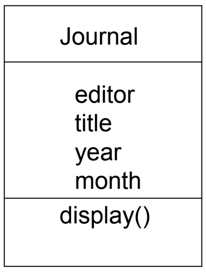


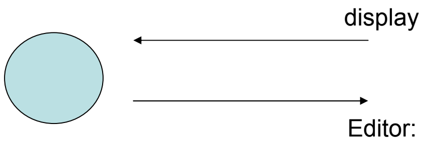
```
public class Journal {

  String editor;
  String title;
  String year;
  String month;

  Journal (String editor, String title, String year, String month) {
    this.editor = editor;
    this.title = title;
    this.year = year;
    this.month = month;
  }

  public void display () {
    System.out.println ("Editor: " + editor + "\n" +
        "Title: " + title + "\n" +
        "Year: " + year + "\n" +
        "Month: " + month + "\n");
  }

}
```

21

Creating a Journal object and sending a message

```
public class TestV02 {
  static public void main(String args[]) {

    Journal MyJournal = new Journal ("David Parnas", "Computer Journal", "2003", "November");
    MyJournal.display();

  }
}
```





```
Editor: David Parnas
Title: Computer Journal
Year: 2003
Month: November
```

22

Extending classes: Inheritance relationships

- <u>Inheritance</u> defines a relationship between classes.
- When class C2 <u>inherits from</u> (or <u>extends</u>) class C1, class C2 is called a <u>subclass</u> or an <u>extended class</u> of C1.
- C1 is the <u>superclass</u> of C2.
- Inheritance: a mechanism for reusing the implementation and extending the functionality of superclasses.
- All public and protected members of the superclass are accessible in the extended class

23

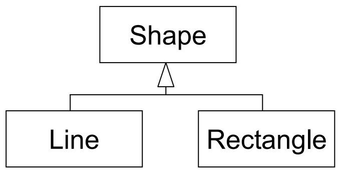
Extending classes: Inheritance relationships (cont.)

- A subclass extends the capability of its superclass.
- The subclass inherits features from its superclass, and may add more features.
- A subclass is a <u>specialization</u> of its superclass.
- <u>Every instance of a subclass is an instance of a superclass, but not vice-versa.</u>

24

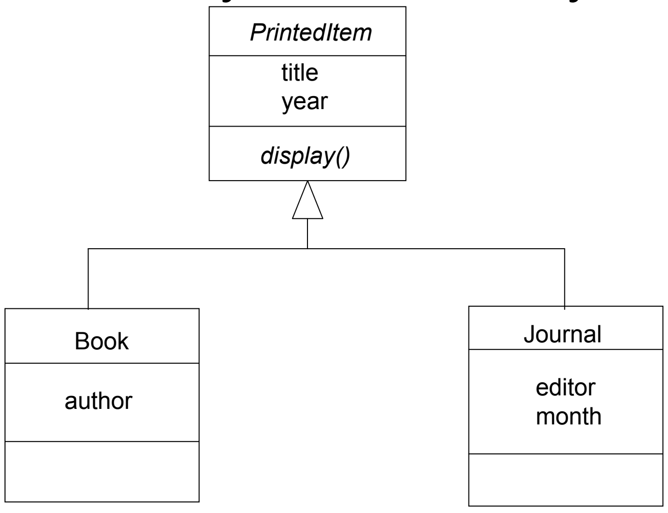
Classes and types

- Each class defines a type. All instances of the class constitute the set of the legitimate values of that type.
- As every instance of a subclass is also an instance of its superclass, the type defined by the subclass is a subset of the type defined by its superclasses.
- The set of all instances of a subclass is included in the set of all instances of its superclass.



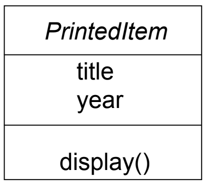
25

Creating an inheritance hierarchy in the library information system



26

Defining a PrintedItem parent class



```
public abstract class PrintedItem {

  String title;
  String year;

  PrintedItem (String title, String year) {
    this.title = title;
    this.year = year;
  }

  public abstract void display ();

}
```

27

Abstract classes

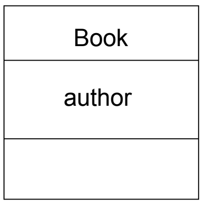
- <u>Abstract classes</u> cannot be directly instantiated.
- Any class that contains abstract methods must be declared abstract.
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or itself be declared abstract.

28

Redefining the Book class: Constructors of extended classes

The keyword *super* refers directly to the constructor of the superclass.

```
public class Book extends PrintedItem {

  String author;

  Book (String author, String title, String year) {
    super(title, year);
    this.author = author;

  }

  public void display () {
    System.out.println ("Author: " + author + "\n" +
        "Title: " + title + "\n" +
        "Year: " + year + "\n");
  }

}
```



29

Redefining the Book class (cont.)

```
public class Book extends PrintedItem {

  String author;

  Book (String author, String title, String year) {
    super(title, year);
    this.author = author;

  }

…

}
```

- The initialization of an extended class consists of two phases:

1. The initialization of the fields inherited from the superclass (one of the constructors of the superclass must be invoked)
2. The initialization of the fields declared in the extended class.

30

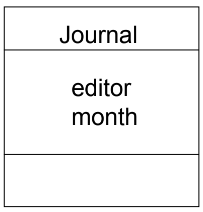
Order of field initialization

```
public class Super {
      int x = …;       // first

      public Super() {
            x = …;    //second
      }
      …
}
```

```
public class Extended extends Super {
      int y = ..;        // third

      public Extended() {
            super();
            y = …;    // fourth
            …
      }
}
```

- The fields of the superclass are initialized, using explicit initializers or the default initial values.
- One of the constructors of the superclass is executed.
- The fields of the extended class are initialized, using explicit initializers or the default initial values.
- One of the constructors of the extended class is executed.

Redefining Journal class



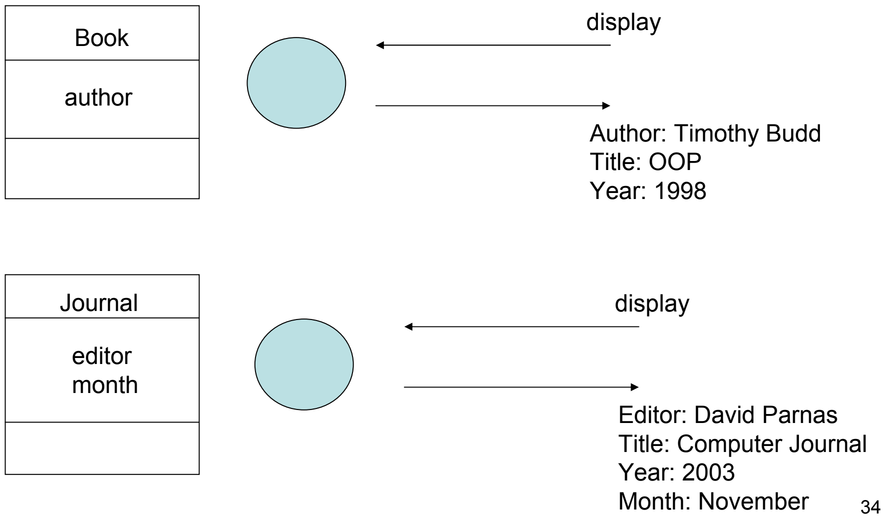
```
public class Journal extends PrintedItem {

  String editor;
  String month;

  Journal (String editor, String title, String year, String month) {
    super(title, year);
    this.editor = editor;
    this.month = month;
  }

  public void display () {
    System.out.println ("Editor: " + editor + "\n" +
        "Title: " + title + "\n" +
        "Year: " + year + "\n" +
        "Month: " + month + "\n");
  }

}
```

Creating a Book and a Journal object and sending a message

```
public class TestV03 {
  static public void main(String args[]) {

    Book MyBook = new Book ("Timothy Budd", "OOP", "1998");

    Journal MyJournal = new Journal ("David Parnas", "Computer Journal",
                                                          "2003", "November");
    MyBook.display();
    MyJournal.display();

  }
}
```

33

Creating a Book and a Journal object and sending a message



Author: Timothy Budd
Title: OOP
Year: 1998

Editor: David Parnas
Title: Computer Journal
Year: 2003
Month: November

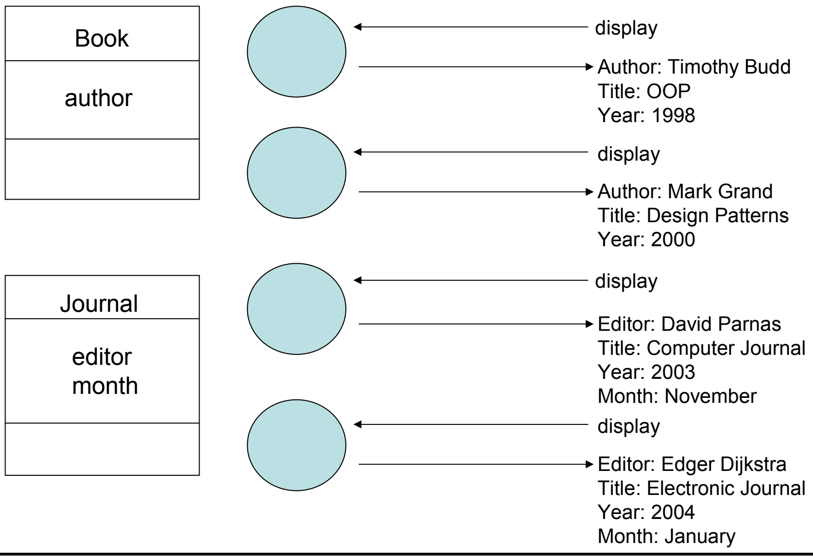
34

Creating Book and Journal objects and sending messages

```
public class TestV04 {
  static public void main(String args[]) {

    Book Book1 = new Book ("Timothy Budd", "OOP", "1998");
    Book Book2 = new Book ("Mark Grand", "Design Patterns", "2000");


    Journal Journal1 = new Journal ("David Parnas", "Computer Journal", "2003", "November");
    Journal Journal2 = new Journal ("Edger Dijkstra", "Electronic Journal", "2004", "January");

    Book1.display();
    Book2.display();

    Journal1.display();
    Journal2.display();

  }
}
```

35

Creating Book and Journal objects and sending messages



36

Another example of abstract class

```
public class Account {
  Account() {this.balance = 0;}

  Account(String name, String account, double balance){
    this.name = name;
    this.account = account;
    this.balance = balance;
  }

  public void getBalance () {System.out.println(balance);}

  public void deposit (double amount) {balance = balance + amount;}

  public void withdraw (double amount) {balance = balance - amount;}

  String name;
  String account;
  double balance;
 }
```

37

Type signatures

- The type signature of a method or constructor is a sequence that consists of types of its parameters.
- Note that the return type, parameter names, and final designations of parameters are not part of the signature.
- Parameter order is significant.

Method	Type signature
String toString()	()
void move(int dx, int dy)	(int, int)
void move(final int dx, final int dy)	(int, int)
void paint(Graphics g)	(Graphics)

38

Method overloading

```
public class Account {

  Account () {
    …
  }

  Account (String name,
             String account,
             double balance) {
    …
  }
…
}
```

- If two methods or constructors in the same class have different type signatures, then they may share the same name; that is, they may be overloaded on the same name.
- The name of the method is said to be overloaded with multiple implementations.

39

Method overloading

```
public class Account {

  Account () {
    …
  }

  Account (String name,
             String account,
             double balance) {
    …
  }
…
}
```

- When an overloaded method is called, the number and the types of the arguments are used to determine the signature of the method that will be invoked.
- Overloading is resolved at compiled time.

40

Instantiating the Account class

```
public class AccountTest {
  static public void main(String args[]) {
    Account a = new Account("bob", "BOB2003", 1000);
    a.deposit(150);
    a.withdraw(50);
    a.getBalance();

    a.withdraw(2000);
    a.getBalance();
  }
}
```

```
1100.0
-900.0
```

41

Introducing SavingsAccount class

```
public abstract class Account {..}
```

```
public class SavingsAccount extends Account {
  SavingsAccount(String name, String account, double balance){
    super(name, account, balance);
  }
  public void withdraw(double amount){
    if (amount > balance)
      System.out.println ("ERROR");
    else
      super.withdraw(amount);
  }
}
```

42

Executing the code

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

    SavingsAccount s = new SavingsAccount ("Joe Smith", "JOESMITH2004", 1000);
    s.getBalance();
    s.withdraw(2000);
    s.getBalance();

  }
}
```

```
1000.0
ERROR
1000.0
```

43

An intuitive description of inheritance

- The behavior and data associated with child classes are always an extension of the properties associated with parent classes.
- A child class will be given all the properties of the parent class, and may in addition define new properties.
- Inheritance is always transitive, so that a class can inherit features from superclasses many levels away.

44

An intuitive description of inheritance (cont.)

- A complicating factor in our intuitive description of inheritance is the fact that subclasses can <u>override</u> behavior inherited from parent classes.

45

Method overriding

- <u>Overriding</u> refers to the introduction of an instance method in a subclass that has the same name, type signature and return type of a method in the superclass.
- The implementation of the method in the subclass <u>replaces</u> the implementation of the method in the superclass.

46

Method overriding (cont.)

```
public class Employee {

  public Employee (String name, double salary) {
    this.name = name;
    this.salary = salary;
  }

  public void display() {
    System.out.println("Name: " + name + " Salary: " + salary + "\n");
  }

  public void raiseSalary (double byPercent) {
    salary = salary + (salary * byPercent / 100);
  }

  private String name;
  private double salary;

}
```

47

Method overriding (cont.)

```
public class Test {
  static public void main(String args[]) {
    Employee e = new Employee ("Janis Joplin", 1000);
    e.display();

    e.raiseSalary(10);
    e.display();
  }
}
```

```
Name: Janis Joplin Salary: 1000.0

Name: Janis Joplin Salary: 1100.0
```

48

Method overriding (cont.)

```
public class Manager extends Employee {

  public Manager (String name, double salary) {
    super(name, salary);
  }

  public void raiseSalary (double byPercent) {
    double bonus = 200;
    super.raiseSalary (byPercent + bonus);
  }

}
```

49

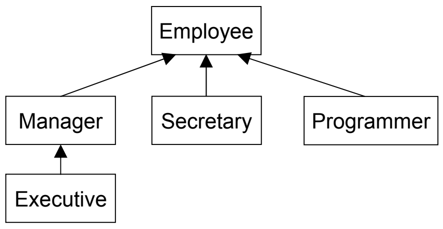
Method overriding (cont.)

```
public class Test {
  static public void main(String args[]) {
    Employee e = new Employee ("Janis Joplin", 1000);
    e.display();

    e.raiseSalary(10);
    e.display();

    Manager m = new Manager ("John Lennon", 1000);
    m.display();
    m.raiseSalary(10);
    m.display();
  }
}
```

```
Name: Janis Joplin Salary: 1000.0
Name: Janis Joplin Salary: 1100.0

Name: John Lennon Salary: 1000.0
Name: John Lennon Salary: 3100.0
```

50

Inheritance hierarchies



- Inheritance does not stop at deriving one layer of classes.
- For example, we can have an Executive class that derives from Manager.

51

Inheritance hierarchies

- The collection of all classes extending from a common parent is called an inheritance hierarchy.

- The path from a particular class to its ancestors in the inheritance hierarchy is its inheritance chain.

52

Overriding and hiding

```
class A {
      int x;
      void y() {...}
      static void z() {...}
}
```

```
class B extends A {
      float x;               // hiding
      void y() {...}         // overriding
      static int z() {...}   // hiding
      }
```

- When a subclass declares a field or static method that is already declared in its superclass, it is not overriden; it is hidden.

53

Overriding versus hiding

- Overriding and hiding are different concepts:
- Instance methods can only be overriden. A method can be overriden only by a method of the same signature and return type.
- When an overriden method is invoked, the implementation that will be executed is chosen at *run time*.
- Static methods and fields can only be hidden. A static method or field may be hidden by a static method or a field of a different signature or type.
- When a hidden method or field is invoked or accessed, the copy that will be used is determined at *compile time*.
- In other words, the static methods and fields are statically bound, based on the declared type of the variables.

54

Overriding versus hiding (cont.)

```
class Point {
       public String className ="Point";
       static public String getDescription() {
              return "Point";
       }
       // other declarations
}
```

```
class ColoredPoint extends Point {
       public String classname = "ColoredPoint";
       static public String getDescription() {
              return "ColoredPoint";
       }
       // other declarations
}
```

Hides field of same name of Point class.

Hides method of same name in Point class.

Overriding versus hiding (cont.)

- Although both p1 and p2 refer to the same object, the binding of the static methods and fields is based on the declared types of the variables at compile time.
- The declared type of p1 is ColoredPoint, and of p2 is Point.

```
ColoredPoint p1 = new ColoredPoint(10.0, 10.0, Color.blue);
Point p2 = p1;

System.out.println(p1.getDescription());
System.out.println(p2.getDescription());
System.out.println(p1.className);
System.out.println(p2.className);
```

```
ColoredPoint
Point
ColoredPoint
Point
```

Object reference with “this”

```
public class Faculty {
        protected Department dept;
        protected String name;
        public Faculty(String n, Department d) {
                name = n; dept = d;
        }
        public Department getDepartment() {
                return dept;
        }...}
```

```
public class Department {
   protected String name;
   protected Faculty facultyList[] = new Faculty[100];
   protected int numOffaculty = 0;
   public Department(String n) {name = n;}
   public void newfaculty(String name) {
      facultyList[numOffaculty++] =
                new faculty(name, this);
   }...}
```

- Keyword `this` is used inside instance methods to refer to the receiving object of the methods, i.e. the object instance through which the method is invoked.
- Keyword `this` may not occur inside static methods.
- The two common uses of this are
  1. to pass the receiving object instance as a parameter
  2. to access instance fields shadowed, or hidden, by local variables.

57

Accessing shadowed fields

```
public class MyClass {
   int var;
   void method1() {
      // local variable shadows instance variable
      float var;
      ...
   }
   void method2(int var) {
      // var also shadows the instance variable
   }
}
```

- A field declared in a class can be shadowed, or hidden, inside a method by a parameter or a local variable of the same name.
- The hidden variable can be accessed as `this.var`

58

Preventing inheritance

- Use the `final` modifier in the class definition to prevent a class from ever becoming a parent class.
  - `final class MyClass {…}`
- You can also make a specific method in a class final in which case no class can override this method.
- All methods in a final class are automatically final.
- The modifier final is the opposite of abstract.
- When applied to a class, it implies that the class can not be subclassified.
- When applied to a method, the keyword indicates that the method cannot be overriden.

59

Subtype relationships

- The subset relation between the value set of types is known as the subtype relationship.
- Type T1 is a subtype of Type T2 if every legitimate value of T1 is also a legitimate value of T2. In this case, T2 is the supertype of T1.
- The inheritance relationship among classes is a subtype relationship. (Also, each interface further defines a type.)
- A value of a subtype can appear wherever a value of a supertype is expected.
  - An instance of a subclass can appear wherever an instance of a superclass is expected.

60

Subtype relationships

- The conversion of a subtype to one of its supertypes is called widening; It is carried out implicitly whenever necessary.
- In other words, a reference to an object of class C can be implicitly converted to a reference to an object of one of the superclasses of C.
- The conversion of a supertype to one of its subtypes is called narrowing.
- Narrowing of reference types requires explicit casts.

61

Abstract classes and subtypes

```
abstract public class AbstractCounter {
        abstract public void click();
        public int get() {return value;}
        public void set(int x) {value = x;}
        public String toString() {
          return String.valueOf(value);
        }
        protected int value;
}
```

```
public class Counter extends AbstractCounter {
   public void click() {
      value =(value + 1) % 100;
   }
}
```

- The class hierarchy that results from the use of inheritance creates a related set of types.
- Instances of any subclass of a given superclass may be referenced by a superclass variable.
- For example, an object of type `Counter` may be referenced by a variable of type `AbstractCounter`.

62

Abstract classes and subtypes

- A subtype can be used anywhere the supertype is expected.
- This principle holds because all the methods needed for the superclass are available for the subclass object.
- The following code fragment shows a method that expects a parameter that is any subclass of `AbstractCounter`:

```
void sampleMethod (AbstractCounter counter) {
        …
        counter.click();
        …
}
```

63

Polymorphic assignments

- The type of the expression at the RHS of an assignment must be a subtype of the type of the variable at the LHS of the assignment.
- For example, if class E extends class B, any instance of E can act as an instance of B.

64

First example of polymorphic assignment

```
class Student {
   public Student (String name) {this.name = name;}
   public String toString() {return "Student: " + name;}
   protected String name;
}
```

```
class Undergraduate extends Student {
      public Undergraduate (String name) {super(name);}
      public String toString() {return "Undergraduate student: " + name;}
}
```

```
class Graduate extends Student {
      public Graduate(String name) {super(name);}
      public String toString() {return "Graduate student: " + name;}
}
```

65

First example of polymorphic assignment (cont.)

```
Student student 1, student2;

student1 = new Undergraduate();          // polymorphic assignment
student2 = new Graduate();               // polymorphic assignment

Graduate student3;
student3 = student2;                     // compilation error;
```

In last statement, RHS not a subtype of LHS.

Type checking is carried out at compile time and it is based on the declared types of variables. An explicit cast is necessary here:

```
Student3 = (Graduate) student2;          // explicit cast; downcasting
```

66

First example of polymorphic assignment (cont.)

```
Student3 = (Graduate) student2;
```

- Validity of explicit cast is checked at run-time. A run-time check will be performed to determine whether `student2` actually holds an object that is an instance of `Graduate` or its subclasses.

```
Student3 = (Graduate) student1;                  // compilation ok
```

- The statement will throw a run-time exception as student1 actually holds an instance of `Undergraduate` (which is not a subtype of `Graduate`)

67

First example of polymorphic assignment (cont.)

- In order to prevent a run-time exception, use the **instanceof** operator:
- The expression `exp instanceof Type` returns a boolean indicating whether `exp` is an instance of a class or an interface named `Type`.

```
if (student1 instanceof Graduate) {
      Graduate gradStudent = (Graduate) student1;
}
else {
      // student1 is not a graduate student
}
```

68

First example of polymorphic assignment (cont.)

Let's assume that the `Graduate` class defines a method `getResearchTopic()` that is not defined in the `Student` class.

```
Student student1 = new Graduate();
// …
student1.getResearchTopic();          // compilation error
```

The declared type of `student1` is `Student`, not `Graduate`, even though `Student1` holds an instance of `Graduate`.

The validity of method invocation is checked statically (at compile time) and it is based on the declared types of variables, not the actual classes of objects.

69

First example of polymorphic assignment (cont.)

Thus, student must be downcast to `Graduate` before invoking `getResearchTopic()`

```
Student student = new Graduate();
…
if (student instanceof Graduate) {
      Graduate gradStudent = (Graduate) student;
      gradStudent.getResearchTopic();
      …
}
…
```

- Why not declare student to be Graduate in the first place?
  - student is a parameter; actual object referred to by student was created in some other part of the program.
  - student is an element of a collection (see later)

70

First example of polymorphic assignment (cont.)

- Instance method toString() is overriden in both subclasses.
- The partivular implementation of toString() cannot be determined at compile time:

```
Student student;
// student is asigned some value
Student.toString();
```

- The implementation to be invoked depends on the actual class of the object referenced by the variable at run-time (and not the declared type of the variable). This is called polymorphic method invocation.

First example of polymorphic assignment (cont.)

- For a polymorphic method invocation

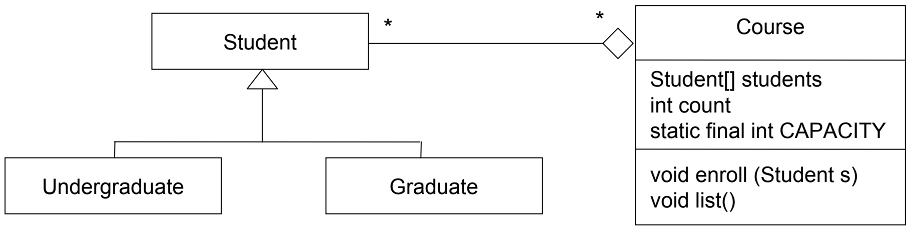
```
var.m();
```

dynamic binding proceeds as follows:

- STEP 1
  - *currentClass* = the class of the object referenced by var.
- STEP 2:

```
IF (method m() is implemented in currentClass)
THEN
    the implementation of m() in currentClass is invoked.
ELSE {
    currentClass = the superclass of currentClass;
    repeat step 2
}
```

Second example on polymorphic assignment



73

Second example on polymorphic assignment (cont.)

```
public class Course {
      public void enroll(Student s) {
            if (s != null && count < CAPACITY)
            students[count++] = s;                        // polymorphic assignment
      }
      public void list() {
            for int i = 0; i < count; i++)
            System.out.println(students[i].toString());  // polymorphic invocation
      }
      protected static final int CAPACITY = 40;
      protected Student students[] = new Student[CAPACITY];
      protected int count = 0;
}
```

74

Second example on polymorphic assignment (cont.)

```
Cource c = new Course();
c.enroll(new Undergraduate(“John”);
c.enroll(new Graduate(“Mark”);
c.enroll(new Undergraduate(“Jane”);
c.list();
```

```
Undergraduate student: John
Graduate student: Mark
Undergraduate student: Jane
```

75

More on polymorphism

- What happens when you send a message to a subclass?
- The subclass checks whether or not it has a method with that name and with exactly the same parameters.
- If so, it uses it.
- If not, the parent class becomes responsible for handling the message and looks for a method with that name and those parameters. If so, it calls that method.

- This message handling can continue moving up in the inheritance chain until a matching method is found or until the inheritance chain is exhausted.

76

Dynamic method dispatch

- When you call a method using the dot operator on an object reference, the declared type of the object reference is checked at compile time to make sure that the method you are calling exists in the declared class.
- At runtime, the object reference could be referring to an instance of some subclass of the declared reference type.
- In these cases, Java uses the actual instance to decide which method to call in the event that the subclass overrides the method being called.

77

Dynamic method dispatch example

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

```
class B extends A {
            void callme() {
               System.out.println("Inside B");
            }
}
```

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

- Java compiler: has to verify that indeed A has a method named `callme()`
- Java runtime: notices that the reference is actually an instance of B, so it calls B's `callme()` method instead of A's.

- The output is "Inside B"

78

Implementing interfaces

```
interface MyInterface {
      // an abstract method
      void aMethod (int i);
}
```

```
class MyClass **implements** MyInterface {
      public void aMethod(int i) {…}
}
```

- Interfaces declare features but provide no implementation.
- An interface encapsulates abstract methods and constants.
- Interface methods cannot be static.
- An interface can extend other interfaces (not classes).
- Classes that implement an interface should provide implementation for all features (methods) declared in the interface.
- Java allows only single inheritance for class extension but multiple inheritance for interface extension.

79

Interfaces and types

- Each interface defines a type.
- The interface extension and implementation are also subtype relations.
- Let us define the complete subtype relations in Java:
  – If class C1 extends class C2, then C1 is a subtype of C2.
  – If interface I1 extends interface I2, then I1 is a subtype of I2.
  – If class C implements interface I, then C is a subtype of I.
  – For every interface I, I is a subtype of Object (the parent class of the entire Java class hierarchy).
  – For every type T (reference or primitive type) T[] is a subtype of Object.
  – If type T1 is a subtype of T2, then T1[] is a subtype of T2[].

80

A first example using interfaces

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

```
class Client implements Callback {
        void callback(int p) {
          System.out.println(“callback called +
             with “ + p);
        }
        void callme() {
          System.out.println(“Inside client”);
        }
}
```

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

- You can declare variables as object references which use an interface as the type rather than a class.
- Any instance of any class which implements the declared interface may be stored in such a variable.
- Variable `c` was declared to be of interface `Callback`, yet it was assigned an instance of `Client`.
- This way, `c` can only be used to access the callback method, and not any of the other aspects of the `Client` class.

81

A second example using interfaces

- Implementing multiple interfaces allows a class to assume different roles in different contexts.

```
interface Student {
   float getGPA();
   // …other methods
}
```

```
interface Employee {
   float getSalary();
   // … other methods
}
```

```
public class FullTimeStudent
            implements Student {
   public float getGPA() {
      // calculate GPA;
   }
   protected float gpa;
   // ..other methods and fields
}
```

```
public class FullTimeEmployee
            implements Employee {
   public float getSalary() {
      // calculate salary;
   }
   protected float salary;
   // ..other methods and fields
}
```

82

A second example using interfaces (cont.)

- The `StudentEmployee` class is a subtype of both Student and Employee.
- Instances of `StudentEmployee` can be treated either as students or as employees.

```
public class StudentEmployee implements Student, Employee {
   public float getGPA() {
      // calculate GPA;
   }
   public float getSalary() {
      // calculate salary;
   }
   protected float gpa;
   protected float salary;
   // ..other methods and fields
}
```

83

A second example using interfaces (cont.)

- In one context, a student employee can be viewed as a student:

```
Student[] students = new Student[…];
students[0] = new FullTimeStudent();
students[1] = new StudentEmployee(); // student employee as a student
// …
for (int i = 0; i < students.length; i++) {
.. Students[i].getGPA()...
}
```

84

A second example using interfaces (cont.)

- In the other context, a student employee can be viewed as an employee:

```
Employee[] employees = new Employee[…];
employees[0] = new FullTimeEmployee();
employees[1] = new StudentEmployee();    // student employee as a employee
// …
for (int i = 0; i < employees.length; i++) {
.. employees[i].getSalary()...
}
```

85

A second example using interfaces (cont.)

```
public class StudentImpl
                    implements Student {
  public float getGPA() {
    // calculate GPA;
  }
  protected float gpa;
}
```

```
public class EmployeeImpl
                   implements Employee {
  public float getSalary() {
    // calculate salary;
  }
  protected float salary;
}
```

- The implementation in `StudentImpl` and `EmployeeImpl` can be directly reused in the full-time student and employee classes by class extension:

```
public class FulltimeStudent
                    extends StudentImpl{…}
public class FulltimeEmployee
                    extends EmployeeImpl{…}
```

86

Delegation

- In addition, a student employee class can be implemented as follows to reuse the implementation in `StudentImpl` and `EmployeeImpl`:

```
public class StudentEmployee implements Student, Employee {
    public StudentEmployee() {
        studentImpl = new StudentImpl();
        employeeImpl = new EmployeeImpl();
        // ...
    }
    public float getGPA() {
        return studentimpl.getGPA();                // delegation
    }
    public float getSalary() {
        return employeeImpl.getSalary();            // delegation
    }
    protected StudentImpl studentImpl;
    protected EmployeeImpl employeeImpl;
    // ..other methods and fields}
```

87

Delegation (cont.)

- The implementation technique used in the `getGPA()` and `getSalary()` methods is known as <u>delegation</u>.
- As the name suggests, delegation implies that the method simply delegates the task to another object, `studentImpl` and `employeeImpl`, respectively.
- The implementation on the `StudentImpl` and `EmployeeImpl` classes is reused through delegation.

```
public float getGPA() {
    return studentimpl.getGPA();            // delegation
}
public float getSalary() {
    return employeeImpl.getSalary();        // delegation
}
```

88

Resolving name conflicts among interfaces

- Names inherited from one interface may collide with names inherited from another interface or class.
- How do we resolve name collisions? If two methods have the same name, then one of the following is true:
  - If they have different signatures, they are overloaded.
  - If they have the same signature and the same return type, they are considered to be the same method.
  - If they have the same signature but different return types, a compilation error will occur.
  - If they have the same signature and the same return type but throw different exceptions, they are considered to be the same method, and the resulting throws list is the union of the two throws lists.

Resolving name conflicts among interfaces (cont.)

```
interface X {
      void method1(int i);
      void method2(int i);
      void method3(int i);
      void method4(int i) throws Exception1;
}
```

```
interface Y {
      void method1(double d);
      void method2(int i);
      int method3(int i);
      void method4(int i) throws exception2;
}
```

Same type signature, different types; compilation error

Overloaded methods

```
public class MyClass implements X, Y {
      void method1(int i) {...}                  // overrides method1 in X
      void method1(double d) {...}               // overrides method1 in Y
      void method2(int i) {...}                  // overrides method2 in X and Y
      void method4(int i)                        // overrides method4 in X and Y
            throws Exception1, Exception2 {...}
}
```

Constants in interfaces

- Two constants having the same name is always allowed, as they are considered to be two separate constants

```
interface X {
      static final int a = …;
}
```

```
interface Y {
      static final double a = …;
}
```

```
public class MyClass implements X, Y {
      void aMethod() {
      …X.a…                  // the constant in X
      …Y.a…                  // the constant in Y
      }
}
```

91

Forms of inheritance

1. Specialization
   – The new class is a specialized variety of the parent class.
   – It satisfies the specifications of the parent class in all relevant aspects.
   – This form always creates a subtype.
   – The most common use of inheritance.

92

Forms of inheritance (cont.)

2. Specification
   - Use of inheritance to guarantee that classes maintain a certain common interface.
   - Child implements the methods described but not implemented in the parent.
   - Subclass is a realization of an incomplete abstract specification (parent class defines the operation but has no implementation).
   - Two different mechanisms to support inheritance of specification:
     1. Through interfaces
     2. Through inheritance of abstract classes

93

Forms of inheritance (cont.)

3. Construction
   - A class inherits almost all of its desired functionality from a parent class, even if there is no logical relationship between the concepts of parent and child class.
     - For example, the concept of a stack and the concept of a vector have little in common;
     - However, from a pragmatic point of view using the vector class as a parent greatly simplifies the implementation of a stack.
   - Principle of substitutability does not always hold; subclasses are not always subtypes.

94

Forms of inheritance (cont.)

4. Extension
   - A child class only adds new behavior to the parent class and does not modify or alter any of the inherited attributes.
   - As the functionality of the parent class remains available and untouched, the principle of substitutability holds and subclasses are always subtypes.

95

Forms of inheritance (cont.)

5. Limitation
   - The behavior of the subclass is smaller or more restrictive than the behavior of the parent class.
   - For example, you can create the class Set in a fashion similar to the way the class Stack is subclassed from vector.
   - You have to ensure that only Set operations are used on the set, and not vector operations. One way to accomplish this would be to override the undesired methods to generate errors.
   - Subclasses are not subtypes.

96

Forms of inheritance (cont.)

6. Combination
   - When discussing abstract concepts it is common to form a new abstraction by combining features of two or more abstractions.
   - The ability of a class to inherit from two or more parent classes is known as multiple inheritance.
   - In Java, this is accomplished through a class to extend an existing class and implement an interface.
   - It is also possible for classes to implement more than one interface.

97

Example: Inheritance vs. Composition

- Inheritance describes an *is-a* relation.

```
      class Stack extends Vector {
           public Object push (Object item) {
            addElement(item); return item;
           }
           public Object top() {
            return elementAt(size()-1);
           }
           public Object pop() {
            Object obj = top();
            removeElementAt(size()-1);
            return obj;
           } }
```

98

- Composition describes a *has-a* relationship.

```
class Stack {
     private Vector theData;
     public Stack() {theData = new Vector();}
     public boolean empty() {return theData.isEmpty();}
     public Object push(Object item) {
       theData.addElement(item); return item;}
     public Object top() {return thedata.lastElement();}
     public Object pop() {
      Object result = theData.lastElement();
      theData.removeElementAt(theData.size()-1);
      return result;
     } }
```

99

Composition and inheritance contrasted: Substitutability

- Inheritance: Classes formed with inheritance are assumed to be subtypes of the parent class.
  - As a result, subclasses are candidates for values to be used when an instance of the parent is expected.
- Composition: No assumption of substitutability is present.
  - With composition, the data types Stack and Vector are entirely distinct and neither can be substituted in situations where the other is required.

100

Composition and inheritance contrasted: Ease of use

- Inheritance
  - The operations of the new data structure are a superset of the operations of the original data structure on which the new object is built.
  - To know exactly what operations are legal for the new structure, the programmer must examine the declaration for the original.
  - To understand such a class (Stack), the programmer must frequently flip back and forth between declarations.
  - For this reason, implementations using inheritance are usually much shorter in code than are implementations constructed with composition

101

- Composition: Simpler than inheritance.
  - It more clearly indicates exactly what operations can be performed on a particular data structure.
  - Looking at the Stack data abstraction with composition, it is clear that the only operations provided for the data type are the test for emptiness, push, top and pop.
  - This is true regardless of what operations are defined for vectors.

102

Composition and inheritance contrasted: Semantics

- Inheritance: It does not prevent the users from manipulating the new structure using methods from the parent class even if these are not appropriate.
  - For example, nothing prevents a Stack user from adding new elements using `insertElementAt()` which would be semantically illegal for the Stack data structure.

Composition and inheritance contrasted: Information hiding

- Inheritance: A component constructed using inheritance has access to fields and methods in the parent class that have been declared as public or protected.

- Composition: A component constructed using composition can only access the public portions of the included component.
  - In the example, the fact that the class Vector is used is an implementation detail.
  - It would be easy to re-implement the class to make use of a different technique (such as a linked list) with minimal impact on the users of the Stack abstraction.
  - If users counted on the fact that a Stack is merely is specialized form of Vector, such changes would be more difficult to implement.

References

- Timothy Budd, *"Understanding Object-Oriented Programming with Java (Updated Edition)",* Addison-Wesley, 2000.
- Xiaoping Jia, *"Object-Oriented Software Development Using Java",* Addison-Wesley, 2000.