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
```

- All method parameters are passed by value (i.e. modifications to parameters of primitive types are made on copies of the actual parameters).
- Objects are passed by reference.
- 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 type signature of a method (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.
- Static blocks: 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

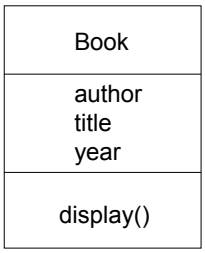
Example on accessing static features

- 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");
  }

}
```

17

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).

18

Sending messages

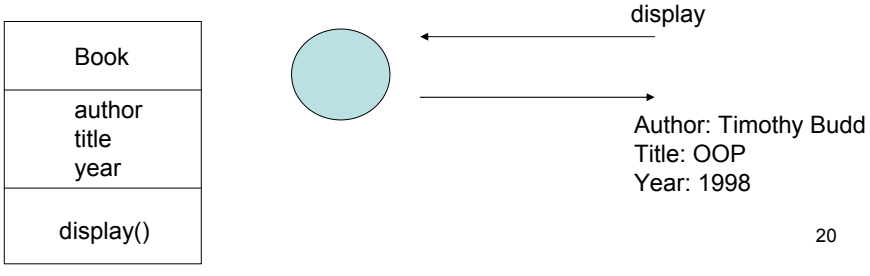
```
public class TestV01 {
  static public void main(String args[]) {

    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

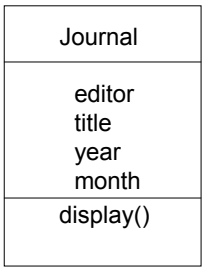
Sending a message to a Book instance

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

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

  }
}
```



20

Defining a Journal class

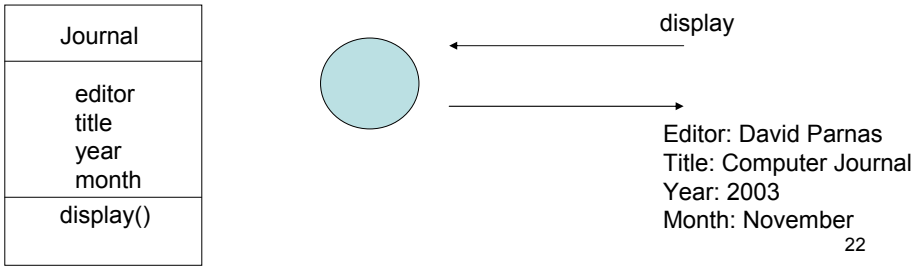



```
public class Journal {

  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();

  }
}
```



22

Extending classes: Inheritance relationships

- Inheritance defines a relationship between classes.
- When class C2 inherits from (or extends) class C1, class C2 is called a subclass or an extended class of C1.
- C1 is the superclass 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

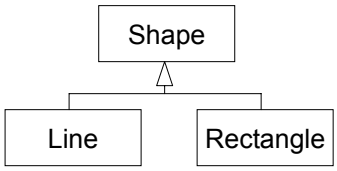
Extending classes: Inheritance relationships (cont.)

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

24

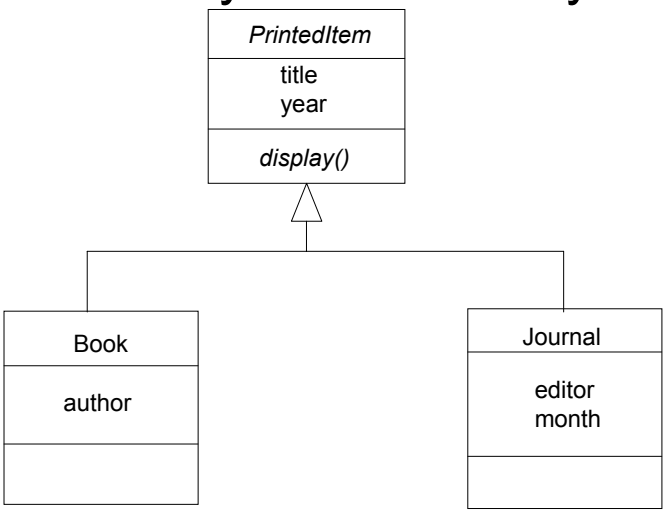
Classes and types

- 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.



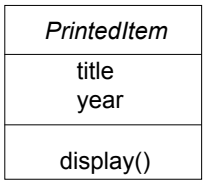
25

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

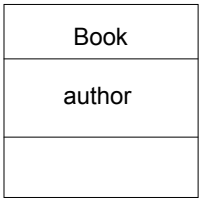
- Abstract classes cannot be directly instantiated.
- Any class that contains abstract methods must be declared abstract.
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or itself be declared abstract.

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");
  }

}
```



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.

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.

31

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");
  }

}
```

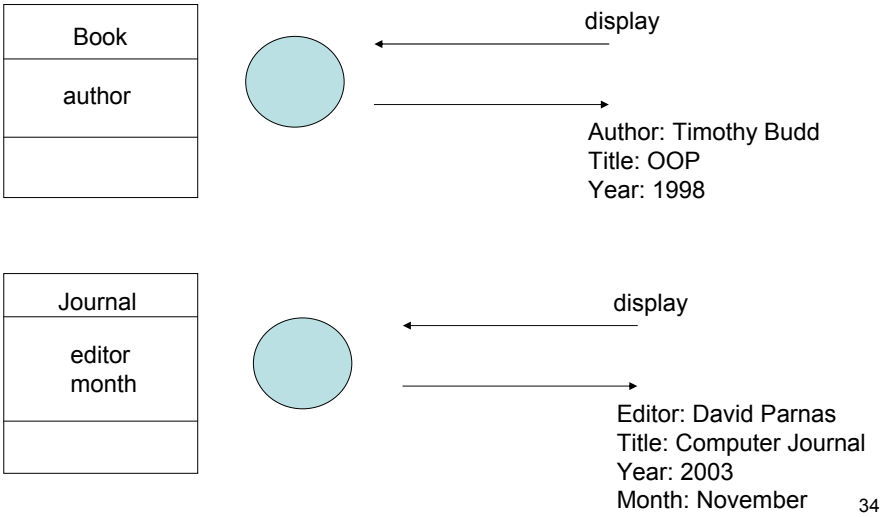
32

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



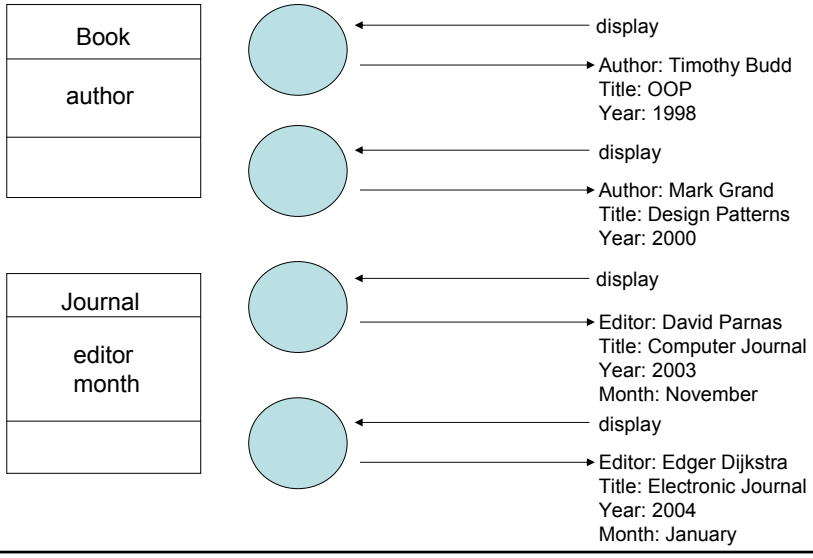
34

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 override behavior inherited from parent classes.

45

Method overriding

- Overriding 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 replaces 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.)

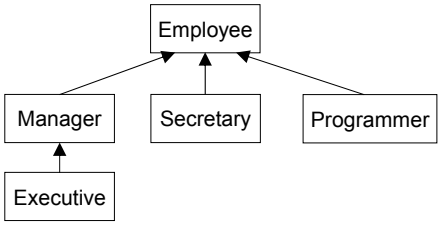
```
public class Manager extends Employee {

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

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

}
```


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
```

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