

COMP9024: Data Structures and Algorithms

Week One: Java Programming
Language (II)

Hui Wu

Session 1, 2015

<http://www.cse.unsw.edu.au/~cs9024>

1

Outline

- ♦ Inheritance and Polymorphism
- ♦ Interfaces and Abstract Classes
- ♦ Exceptions
- ♦ Casting and Generics

2

Object-Oriented Design Goals

- ♦ Robustness
 - Capable of handling unexpected inputs that are not explicitly defined for an application.
- ♦ Adaptability
 - Running a software with minimal changes on different hardware and operating system platforms.
- ♦ Reusability
 - The same code is usable as a component of different systems in various applications.

3

Object-Oriented Design Principles

- ♦ Abstraction
 - Distill a complicated system down to its most fundamental parts and describe these parts in a simple, precise language.
- ♦ Encapsulation
 - Different components of a software system should not reveal the internal details of their respective implementations.
- ♦ Modularity
 - Different components of a software system are divided into separated functional units.

4

Inheritance

- ♦ A technique that allows the design of general classes that can be specialized to more particular classes, with the specialized classes reusing the code from the general class.
- ♦ A class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*). The class from which the subclass is derived is called a *superclass* (also a *base class* or a *parent class*).

5

What a Subclass Inherits from Its Superclass?

- ♦ A subclass inherits all the *non-private members* (fields, methods, and nested classes) from its superclass.
- ♦ Constructors are *not* inherited by subclasses, but the constructor of the superclass can be invoked from the subclass by using the operator *super*.

6

An Example of Inheritance (1/2)

```
public class Bicycle { // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed; // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed; } // the Bicycle class has four methods
    public void setCadence(int newValue) { cadence = newValue; }
    public void setGear(int newValue) { gear = newValue; }
    public void applyBrake(int decrement) { speed -= decrement; }
    public void speedUp(int increment) { speed += increment; } }
```

7

An Example of Inheritance (2/2)

```
public class MountainBike extends Bicycle {
    // the MountainBike subclass adds one field
    public int seatHeight;
    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight, int startCadence, int startSpeed,
        int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight; }
    // the MountainBike subclass adds one method
    public void setHeight(int newValue) { seatHeight = newValue; } }
```

8

What Can You Do in a Subclass? (1/2)

- ♦ The inherited fields can be used directly, just like any other fields.
- ♦ You can declare a field in the subclass with the same name as the one in the superclass, thus *hiding* it (not recommended).
- ♦ You can declare new fields in the subclass that are not in the superclass.
- ♦ The inherited methods can be used directly as they are.

9

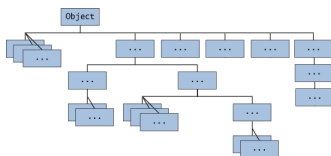
What Can You Do in a Subclass? (2/2)

- ♦ You can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus *overriding* it.
- ♦ You can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus *hiding* it.
- ♦ You can declare new methods in the subclass that are not in the superclass.
- ♦ You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword *super*.

10

Java Class Hierarchy

- ♦ Excepting Object, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object.



11

Polymorphism

- ♦ The ability of an object variable to take different forms.
 - Overriding
 - If a subclass and its superclass contain two instance methods with the same name, the instance method in the subclass overrides the one with the same name in its superclass.
 - Hiding
 - If a subclass and its superclass contain two class methods with the same name, the class method in the subclass hides the one with the same name in its superclass.
 - Overloading
 - A single class may contain multiple methods with the same name but different signatures. The signature of a method is a combination of its name and the type and number of arguments that are passed to it.

12

An Example of Overriding and Hiding (1/2)

```
public class Animal {  
    public static void testClassMethod() {  
        System.out.println("The class method in Animal.");  
    }  
  
    public void testInstanceMethod() {  
        System.out.println("The instance method in Animal.");  
    }  
}
```

13

An Example of Overriding and Hiding (2/2)

```
public class Cat extends Animal {  
    public static void testClassMethod() {  
        System.out.println("The class method in Cat.");  
    }  
    public void testInstanceMethod() {  
        System.out.println("The instance method in Cat.");  
    }  
    public static void main(String[] args) {  
        Cat myCat = new Cat();  
        Animal myAnimal = myCat;  
        Animal.testClassMethod();  
        myAnimal.testInstanceMethod();  
    }  
}
```

14

Summary of Overriding and Hiding

Defining a Method with the Same Signature as a Superclass's Method:

	Superclass Static Method	Superclass Instance Method
Subclass Static Method	Hide	Generates a compile-time error
Subclass Instance Method	Generates a compile-time error	Override

15

Dynamic Dispatch

- When a program invokes a certain method `a()` of an object `o`, the Java run-time environment checks if `a()` is defined in `o`'s class `T`. If yes, then executes it; otherwise, checks if it is defined in `T`'s superclass `S`. If yes, then executes it; otherwise, checks if it is defined in `S`'s superclass `P`. This procedure is repeated until a definition of this method is found, or the topmost class is reached, in which case the Java runtime environment will generate a runtime error.

16

Accessing Superclass Members (1/2)

- A subclass can access the members of its superclass via the keyword **super**.
- Consider the following example:

```
public class Superclass {  
    public void printMethod() {  
        System.out.println("Printed in Superclass.");  
    }  
}
```

17

Accessing Superclass Members (2/2)

```
public class Subclass extends Superclass {  
    public void printMethod() {  
        //overrides printMethod in Superclass  
        super.printMethod();  
        System.out.println("Printed in Subclass");  
    }  
    public static void main(String[] args) {  
        Subclass s = new Subclass();  
        s.printMethod();  
    }  
}
```

18

Subclass Constructors (1/2)

- The syntax for calling a superclass constructor is

```
super();  
or  
super(parameter list);
```

- An example:

```
public MountainBike(int startHeight, int startCadence,  
int startSpeed, int startGear) {  
    super(startCadence, startSpeed, startGear);  
    seatHeight = startHeight; }  
}
```

19

Subclass Constructors (2/2)


- Invocation of a superclass constructor must be the first line in the subclass constructor.
- If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the super class does not have a no-argument constructor, you will get a compile-time error.

20

Referencing the Current Instance of a Class (1/2)

- Java provides the **this** operator to reference the current instance of a class.

```
public class Point {  
    public int x = 0;  
    public int y = 0;  
    //constructor  
    public Point(int a, int b) {  
        x = a;  
        y = b; }  
}
```



```
public class Point {  
    public int x = 0;  
    public int y = 0;  
    //constructor  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y; }  
}
```

21

Referencing the Current Instance of a class (2/2)

```
public class Rectangle {  
    private int x, y, width, height;  
    public Rectangle() { this(0, 0, 0, 0); }  
    public Rectangle(int width, int height) {  
        this(0, 0, width, height); }  
    public Rectangle(int x, int y, int width, int height) {  
        this.x = x;  
        this.y = y;  
        this.width = width;  
        this.height = height; } ... }  
}
```

22

Exceptions (1/5)

- An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred.
- Creating an exception object and handing it to the runtime system is called *throwing an exception*.

23

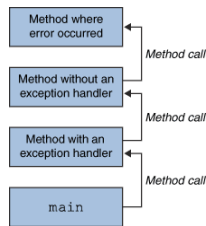
Exceptions (2/2)

- Locating the exception handler:
 - The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an *exception handler*.
 - The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler.
 - An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

24

Exceptions (3/2)

- ♦ The call stack:



25

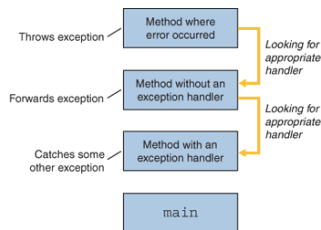
Exceptions (4/2)

- ♦ Searching the call stack for the exception handler:
 - The exception handler chosen is said to *catch the exception*.
 - If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the runtime system (and, consequently, the program) terminates.

26

Exceptions (5/2)

- ♦ Searching the call stack for the exception handler:



27

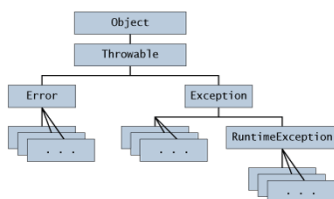
Three Types of Exceptions

- ♦ Checked exception
 - These are exceptional conditions that a well-written application should anticipate and recover from.
- ♦ Error
 - These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from.
- ♦ Runtime exception
 - These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from.

28

The Throwable Class

- ♦ Java platform provides numerous exception classes. All the classes are descendants of the **Throwable** class.



29

Error and Exception Classes

- ♦ Error class
 - When a dynamic linking failure or other hard failure in the Java virtual machine occurs, the virtual machine throws an Error.
- ♦ Exception class
 - Most programs throw and catch objects that derive from the Exception class. An Exception indicates that a problem occurred, but it is not a serious system problem.

30

Throwing the Exceptions

- ♦ Syntax:

`throw someThrowableObject`

- ♦ An example:

```
public Object pop() {
    Object obj;
    if (size == 0) {
        throw new EmptyStackException();
    }
    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

31

Specifying the Exceptions

- ♦ When a method is declared, it is appropriate to specify the exceptions it might throw by using the keyword **throws**.

- ♦ An example:

```
public void getReadyForClass() throws
    ShoppingListTooSmallException, OutOfMoneyException {
    goShopping(); // I don't have to try or catch the exceptions
                // which goShopping() might throw because
                // getReadyForClass() will just pass these along.
    makeCookiesForTA();
}
```

32

Catching Exceptions (1/2)

- ♦ Syntax:

```
try
    main block of statement
catch (exceptionType1 variable)
    block 1 of statements
catch (exceptionType2 variable)
    block 2 of statements
...
finally
    block n of statements
```

33

Catching Exceptions (1/2)

- ♦ Syntax:

```
try
    main block of statements
catch (exceptionType1 variable1)
    block 1 of statements
catch (exceptionType2 variable2)
    block 2 of statements
...
finally
    block n of statements
```

34

Catching Exceptions (2/2)

```
public void writeList() { PrintWriter out = null;
try { System.out.println("Entering try statement");
    out = new PrintWriter( new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++)
        out.println("Value at: " + i + " = " + vector.elementAt(i));
catch (ArrayIndexOutOfBoundsException e) {
    System.err.println("Caught " + "ArrayIndexOutOfBoundsException: " +
        e.getMessage());
catch (IOException e) { System.err.println("Caught IOException: " +
        e.getMessage());
finally {
    if (out != null) { System.out.println("Closing PrintWriter");
        out.close();
    }
    else { System.out.println("PrintWriter not open"); } } }
```

35

Interface (1/7)

- ♦ A Java interface is a collection of methods declarations without data and bodies.
- ♦ An interface cannot be instantiated—it can only be implemented by a class or extended by another interface.

36

Interface (2/7)

- Defining an interface is similar to creating a new class:

```
/** Interface for objects that can be sold. */
public interface Sellable { /** description of the interface */
    public String description(); /** list price in cents */
    public int listPrice(); /** lowest price in cents we will accept */
    public int lowestPrice(); }
```

37

Interface (3/7)

- To use an interface, you write a class that *implements* the interface:

```
/** Class for photographs that can be sold */
public class Photograph implements Sellable {
    private String desc; // description of this photo
    private int price; // the price we are setting
    private boolean color; // true if photo is in color
    public Photograph(String desc, int p, boolean c) { // constructor
        desc = desc; price = p; color = c; }
    public String description() { return desc; }
    public int listPrice() { return price; }
    public int lowestPrice() { return price/2; }
    public boolean isColor() { return color; } }
```

38

Interface (4/7)

- A class can implement multiple interfaces.
- Consider the following example:

```
/** Interface for objects that can be transported. */
public interface Transportable { /** weight in grams */
    public int weight(); /** whether the object is
    hazardous */
    public boolean isHazardous(); }
```

39

Interface (5/7)

- The following class implements both `Sellable` and `Transportable`:

```
/** Class for objects that can be sold, packed, and shipped. */
public class BoxedItem implements Sellable, Transportable
{
    private String desc; // description of this item
    private int price; // list price in cents
    private int weight;
    // weight in grams
    private boolean haz;
    // true if object is hazardous
    private int height=0; // box height in centimeters
```

40

Interface (6/7)

```
private int width=0; // box width in centimeters
private int depth=0; // box depth in centimeters
/** Constructor */
public BoxedItem(String desc, int p, int w, boolean h) {
    desc = desc; price = p; weight = w; haz = h; }
public String description() { return desc; }
public int listPrice() { return price; }
public int lowestPrice() { return price/2; }
public int weight() { return weight; }
public boolean isHazardous() { return haz; }
public int insuredValue() { return price*2; }
public void setBox(int h, int w, int d) {
    height = h; width = w; depth = d; }
```

41

Interface (7/7)

- Although Java does not allow *multiple inheritance* for classes, it allows multiple inheritance for interfaces:

```
public interface InsurableItem extends Transportable,
    Sellable {
    /** Returns insured Value in cents */
    public int insuredValue(); }
```

```
public class BoxedItem2 implements InsurableItem {
    // ... same code as class BoxedItem }
```

42

Abstract Classes

- An abstract class is a class that contains empty method declarations (that is, declarations of methods without bodies) as well as concrete definitions of methods and/or instance variables.
- Like an interface, an abstract class may not be instantiated. A subclass must provide an implementation for the abstract methods of its superclass, unless it is itself abstract.

43

Type Conversions (1/3)

- Widening conversions
 - A widening conversion occurs when a type T is converted into a "wider" type U. Common cases:
 - T and U are class type and U is the superclass of T.
 - T and U are interface types and U is the superinterface of T.
 - T is a class that implements interface U.
- Narrowing conversions
 - A narrowing conversion occurs when a type T is converted into a "narrower" type S. Common cases:
 - T and S are class type and S is the subclass of T.
 - T and S are interface types and S is the subinterface of T.
 - T is the interface that the class S implements.

44

Type Conversions (2/3)

- Widening conversions are automatically performed by Java to store the results of an expression into a variable. Therefore, no cast is needed.
- An example:

```
Integer i = new Integer(3);
Number n = i; // widening conversion from Integer to Number
```

45

Type Conversions (3/3)

- In general narrowing conversions require an explicit cast.
- An example:

```
/** widening conversion from Integer to Number */
Number n = new Integer(2);
/** narrowing conversion from Number to Integer */
Integer i = (Integer) n;
```

Cast

46

The Operator `instanceof`

- Java provides an operator, `instanceof`, that allows us to test if an object variable is referring to an object of a certain class.
- An example:

```
Number n;
Integer i;
n = new Integer(3);
if (n instanceof Integer) i = (Integer) n; // This is legal
n = new Double(3.1415);
if (n instanceof Integer) i = (Integer) n; // This will not be attempted
```

47

Generics (1/4)

- Java provides a generic framework for using abstract types in a way that avoids many explicit casts.
- An generic type is a type that is not defined at compile time, but becomes fully specified at run time.
- The generic framework allows us to define a class in terms of a set of formal type parameters, which could be used, for example, to abstract the types of some internal variables of the class.
- When an object is instantiated, actual type parameters are passed to indicate the concrete types to be used.

48

Generics (2/4)

```
public class Pair<K, V> {
    K key; V value;
    public void set(K k, V v) { key = k; value = v; }
    public K getKey() { return key; }
    public V getValue() { return value; }
    public String toString() { return "[" + getKey() + ", " + getValue() + "]"; }
    public static void main (String[] args) {
        Pair<String, Integer> pair1 = new Pair<String, Integer>();
        pair1.set(new String("height"), new Integer(36));
        System.out.println(pair1);
        Pair<Student, Double> pair2 = new Pair<Student, Double>();
        pair2.set(new Student("A5976", "Sue", 19), new Double(9.5));
        System.out.println(pair2); } }
```

49

Generics (3/4)

- ♦ In the previous example, the actual parameter can be of any type. To restrict the type of an actual parameter, we can use an **extends** clause as shown below:

```
public class PersonPairDirectoryGeneric<P extends Person> {
    /** ... instance variables would go here ... */
    public PersonPairDirectoryGeneric() { /* default constructor goes here */ }
    public void insert (P person, P other) { /* insert code goes here */ }
    public P findOther (P person) { return null; } // stub for find
    public void remove (P person, P other) { /* remove code goes here */ }
}
```

50

Generics (4/4)

- ♦ **Type Parameter Naming Conventions:**
 - By convention, type parameter names are single, uppercase letters. The most commonly used type parameter names are:
 - E - Element (used extensively by the Java Collections Framework)
 - K - Key
 - N - Number
 - T - Type
 - V - Value
 - S, U, V etc. - 2nd, 3rd, 4th types

51

References

1. Chapter 2, Data Structures and Algorithms by Goodrich and Tamassia.
2. The Java™ Tutorials (<http://java.sun.com/docs/books/tutorial/>).

52