300130
Internet Programming

Lecture 2

OOP and Exceptions

2

Main Topics of Lecture

- OOP
- Exception handling

Object Orientation

- Data abstraction & abstract data types
  - classes, abstract classes, interfaces
- Encapsulation and data hiding
- Inheritance and software reusability
  - superclass, subclasses
- Polymorphism

Data Abstraction

- All data abstracted into classes
  - Everything has a type
  - Even the primitives have their underlying classes, e.g. **int.class**, **float.class**

Encapsulation and Data Hiding

- Encapsulate data and methods into objects
- Hide certain data and methods from other objects
  - Private
  - Implementation details hidden within the objects
- Provide only designated data access and interface
  - Communicate with one another across well-defined interfaces

Inheritance

- **New classes are derived as an "extension" of existing classes**
  - Root of all classes is **Object**
  - Each new class must derive from just ***one*** existing class

```
[modifiers] class B extends A { … }
```

- Class B is a *subclass* of class A
- A is *superclass* of B
- New class derives from **Object** if no explicit extends clause exists

Modifiers for Classes

- **public**: available to all
- **abstract**: typically some methods not implemented yet
- **final**: no subclasses can be derived
- Keyword public ***absent***: the class is visible to all package members

Function f() visible to class B

```
class A {  private int a, b;   void f() {}; }
final class B extends A { int c; void g() { f(); } }
```

No more subclasses

Rules of Inheritance

- New subclasses can be derived from any class if it's not **final**
- Subclasses carry all class members of the superclass they derive from (**inheritance**)
- Subclass may possess any new members
- Subclass members may access members of the superclass when **proper permissions** are given
- Inherits also those in the superclasses of the superclasses

Access control: fields

A class variable declaration looks like

> [*AccessSpecifier*] [**static**] [**final**] *VariableType VariableName*

- *AccessSpecifier*: **public**, **protected**, **private**, or non-existent
- **final**: variable is a constant
- **static**: fields belong to the class

Access control: methods

A class method declaration looks like

[*accessSpecifier*] [**static**] [**abstract**]
[**final**] [**synchronized**]
ReturnType MethodName (*ParamList*)
[**throws** *ExceptionList*]

- **final**: method not allowed to override
- **static:** can be called without creating an object of class
- **abstract**: method declared but not defined
- **synchronized**: method synchronized for concurrent programming
- **throws**: throws exceptions

Access Rights

Access	class	subclass	package	world
private	Y			
protected	**Y**	**Y**	**Y**	
public	Y	Y	Y	Y
default	Y		Y	

Final Methods

- A *final* method in a superclass cannot be overridden in a subclass
- Methods that are declared *private* are implicitly *final*
- Methods that are declared *static* are implicitly *final*
- Calls to final methods are resolved at *compile time*—this is known as static binding
- All methods in a final class are implicitly final

Uniquely Numbered Instances

```
public class Id {
  private static int nextId=1;
  private final int  id= nextId++;
  ...
}
```

All instances assigned a unquie ID

- **final** makes the *id* not modifiable
- **id** can be alternatively initiated by a constructor

Inheritance Example

```
class A {
    protected int i;
    public void f(){};
}
```

```
class B extends A {
    private int j;
    protected void g(){};
    private void h(){};
}
```

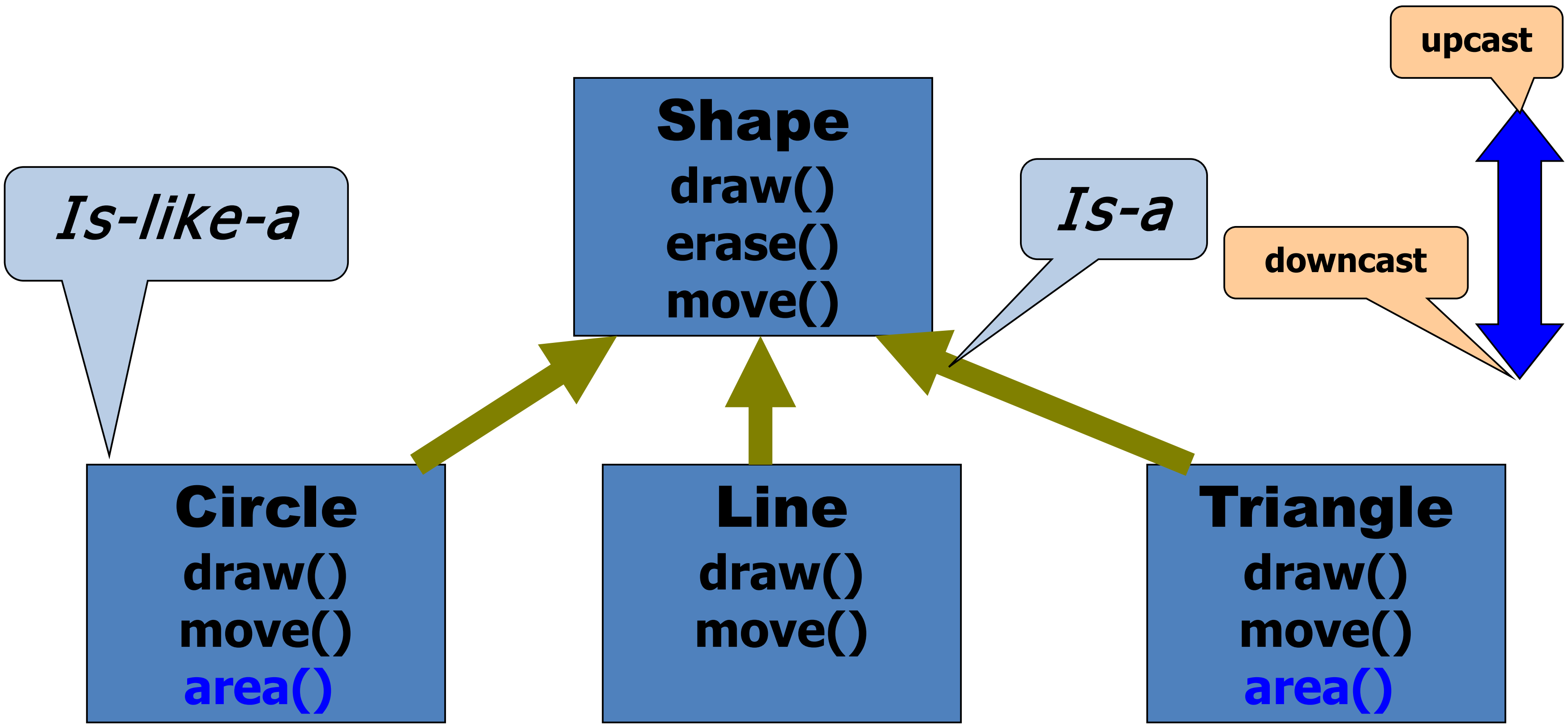
```
class C extends B {
    int k;
    void m() {
        i=1; // ok
        j=2; // illegal
        f(); // ok
        g(); // ok
        h(); // illegal
    }
}
```

//all classes give PACKAGE ACCESS

Polymorphism

- Method has **multi faces/implementations**
- **Subclasses may share the same methods** as the common superclasses
- Actual implementation of that method differs from subclass to subclass
- The method typically *defined* or *overridden* in the subclasses
- Use of same method name **conceptually simplifies the design and maintenance**

Example of Shapes



- Common methods for **Shape** shared
- More methods added or implemented in subclasses
- move() can have different implementations

Polymorphism

- A program invokes a method through a superclass Shape variable
  - Shape myshape
- At execution time, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable
  - myshape=aline;
  - myshape.draw();
- Dynamic binding.

Overriding/Overloading

This f() can't be overridden

```
class A {
 int i=0;
 private void f(){ i=1;};
 void g(){ i=11; };
 A() { f(); }
 A(boolean x) { g(); }
}
```

No overriding

overriding

```
class B  extends A {
 void f(){ i=9; };
 public void g(){ i=99; };
 public int h() { return i;}
 B() { super(); }
 B(boolean x) {super(x); }
}
```

```
class C {
 static B b1=new B(), b2=new B(), b3=new B(true);
 public static void main(String[] args) {
   b2.f();  // overriding: can't see A.f()
   System.out.println(b1.h()+","+b2.h()+","+b3.h());
}}
```

run

Overloaded Constructors

- Enable objects of a class to be initialized in different ways
  - E.g. A( ) and A(boolean x)
- Same name with different signatures

toString method in *any* class:
***object*.toString()** returns a string describing the object

Reusing Classes

- Via composition
  - Create objects of existing classes inside new class
  - All user interface are seen in the new class
- Via inheritance
- Via both composition and inheritance
- Composition more often used

Abstract Class & Methods

- **Normally contain one or more abstract methods**
- ***No*** instances can be made from abstract classes
- An **abstract method** is a method that is not defined yet: only parameters and return type are specified

new A() bad

f( ) not defined

new B() ok

```
abstract class A {
  void f() { }
  abstract float f(int x);
}
```

```
class B extends A {
 float f(int x) { return x; }
}
```

Abstract Classes

- Provides a superclass from which other classes can inherit and thus share a common design
- are *incomplete*
- Constructors and static methods cannot be declared abstract
- Subclasses must declare the “missing pieces” to become “concrete” classes

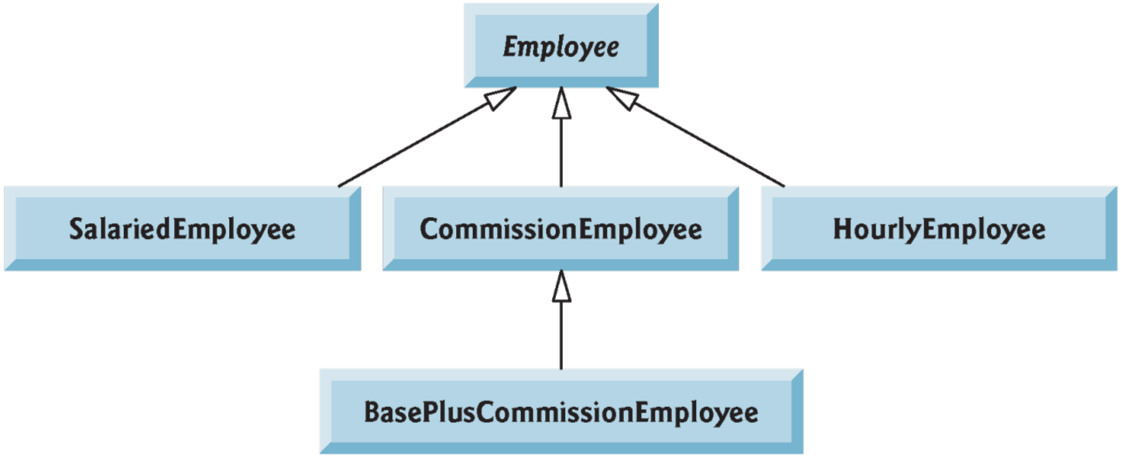


Fig. 10.2 | `Employee` hierarchy UML class diagram.

```java
// Fig. 10.4: Employee.java
// Employee abstract superclass.

public abstract class Employee
{
   private final String firstName;
   private final String lastName;
   private final String socialSecurityNumber;

   // constructor
   public Employee(String firstName, String lastName,
      String socialSecurityNumber)
   {
      this.firstName = firstName;
      this.lastName = lastName;
      this.socialSecurityNumber = socialSecurityNumber;
   }

   // return first name
   public String getFirstName()
   {
      return firstName;
   }

```

Fig. 10.4 | Employee abstract superclass. (Part 1 of 2.)

```java
    // return last name
    public String getLastName()
    {
       return lastName;
    }

    // return social security number
    public String getSocialSecurityNumber()
    {
       return socialSecurityNumber;
    }

    // return String representation of Employee object
    @Override
    public String toString()
    {
       return String.format("%s %s%nsocial security number: %s",
          getFirstName(), getLastName(), getSocialSecurityNumber());
    }

    // abstract method must be overridden by concrete subclasses
    public abstract double earnings(); // no implementation here
 } // end abstract class Employee
```

Fig. 10.4 | Employee abstract superclass. (Part 2 of 2.)

Interface

- An interface is a collection of method signatures, default methods, static methods and constant definitions
- A “purist” form of an abstract class
- May carry nothing but the name
  - Classifying a group of classes that implement it

[**public**] **interface** *InterfaceName*
[**extends** *ListofSuperInterfaces*] **{** … **}**

Interface

- All methods declared in an interface are implicitly public abstract methods
- All fields are **static** and **final**
- Interfaces can have **static** methods with implementations
- Interfaces can have **default** methods with implementation from java 8 onwards

Exceptions

- An exception is an event that disrupts normal flow of execution instructions
  - Caused by hardware failure, software errors, by programming design
- Throwing an exception
  - generate exception info wrapped in an object
  - return to a higher context for solution
- Exception handler ***catches*** the exception

try/catch/finally clause

- Protected code put in a **try block**
- followed by **catch block**(s), the exception handler
- Followed by optional **finally block**
  - Always executed
  - Typically for non-memory cleanups

```
try {
// code prone to exceptions}
catch(Type1  obj1) {
//deals with Type1 exception}
catch(Type2  obj2) {
  // ...}
catch( … ) {
  // ...}
finally {
// code always executed
}
```

throw new ..

Order of *Type1*, *Type2* etc!

Throwing an Exception

- **Exception object created in usual way (**with **new)**
  - **throw new someException( )**
  - Current path of execution then stopped
  - Exception handler then takes over along with passed exception object
- **Advantage of using exceptions**
  - Separating error handling from “regular” code
  - Propagating errors up the call stack
  - Grouping and differentiating error types

throws clause

- Specifies the **exceptions** the method throws
- Passes the exception to a higher level
- Appears **after** the method’s parameter list and **before** the method’s body.
- A method can throw exceptions of the classes listed in its throws clause or of their subclasses

```
Public SomeType someMethod(…)
    throws ExceptionType1, ExceptionType 2 {…}
```

catch block

- Exception handler
- The first catch whose type matches the type of the exception is executed
- Use **System.err**
- Multi-catch
  - catch (Type1 | Type2 | Type3 object)

Example on Using Exceptions

```
class MyEx extends Exception { }
class test {
    static int i=0;
    static void f() throws MyEx {
        if(i==7) throw new MyEx();  }
public static void main(String[] args) {
  do {
        try { f(); }
         catch(MyEx e) {
            System.out.println("i="+i);
            e.printStackTrace();    }
      } while (i++<10);
}}
```

Must derive from **Throwable**

```
i=7
MyEx
at test.f(test.java:6)
at test.main(test.java:10)
```

output

Run

Hierarchy of JAVA Exception

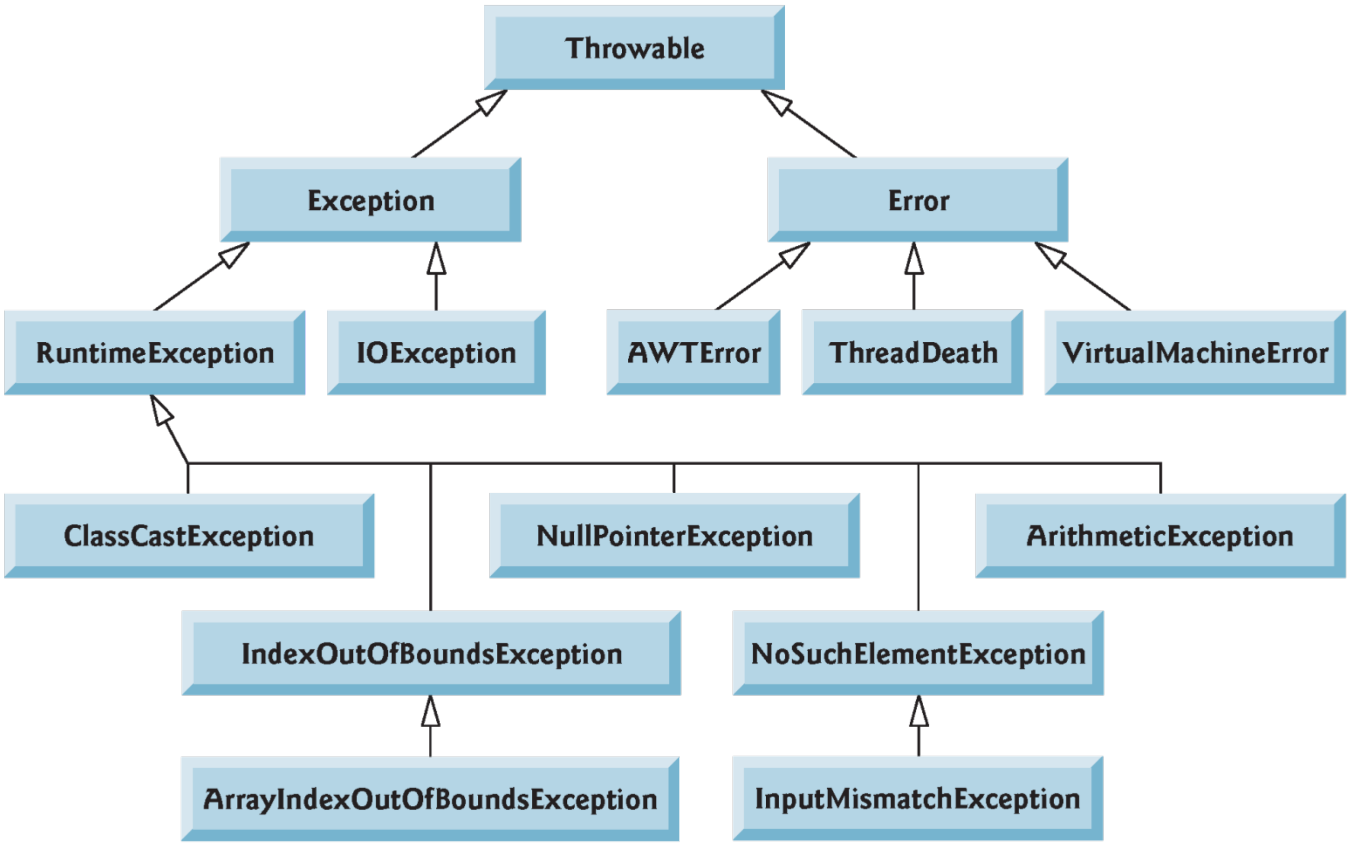


Fig. 11.4 | Portion of class `Throwable`'s inheritance hierarchy.

Error and its subclasses

- Represent abnormal situations that happen in the JVM
- Happen infrequently

unforeseeabe

- Should not be caught by applications
- Applications usually cannot recover from Errors.

Exception and its subclasses

- Represent exceptional situations that can occur in a Java program
- Can be caught and handled by the application
- **Unchecked** Exceptions
  - Subclasses of RuntimeException
  - Dealt with automatically if not caught, **printStackTrace**()
- **Checked** exceptions
  - Subclasses of Exception but not RuntimeException
  - Should be caught or declared in a throws clause
  - Known as the catch-or-declare requirement

Compile-time error, if not

An Example

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class DivideByZeroWithExceptionHandling
{
  // demonstrates throwing an exception when a divide-by-zero occurs
  public static int quotient( int numerator, int denominator )
    throws ArithmeticException
  {
    return numerator / denominator; // possible division by zero
  } // end method quotient
```

Example Cont

```
public static void main( String args[] )
  {
    Scanner scanner = new Scanner( System.in ); // scanner for input
    boolean continueLoop = true; // determines if more input is needed
   do
    {
      try // read two numbers and calculate quotient
      {
        System.out.print( "Please enter an integer numerator: " );
        int numerator = scanner.nextInt();
        System.out.print( "Please enter an integer denominator: " );
        int denominator = scanner.nextInt();
        int result = quotient( numerator, denominator );
        System.out.printf( "\nResult: %d / %d = %d\n", numerator,
          denominator, result );
        continueLoop = false; // input successful; end looping
      } // end try
```

Example Cont

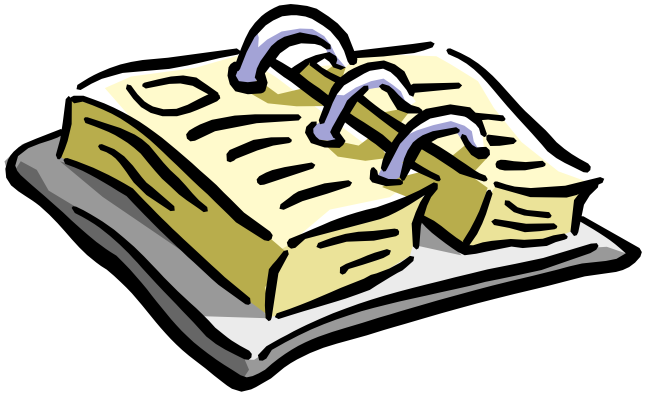
```
 catch ( InputMismatchException inputMismatchException )
{
  System.err.printf( "\nException: %s\n",
    inputMismatchException );
  scanner.nextLine(); // discard input so user can try again
  System.out.println(
    "You must enter integers. Please try again.\n" );
} // end catch
```

Example Cont

```
catch ( ArithmeticException arithmeticException )
      {
        System.err.printf( "\nException: %s\n", arithmeticException );
        System.out.println(
          "Zero is an invalid denominator. Please try again.\n" );
      } // end catch
    } while ( continueLoop ); // end do...while
  } // end main
} // end class DivideByZeroWithExceptionHandling
```

Run

Reading



- **Java How To Program**
  - Chapters 8,9,10,11