



# Object-Oriented Design

# Object Oriented Design

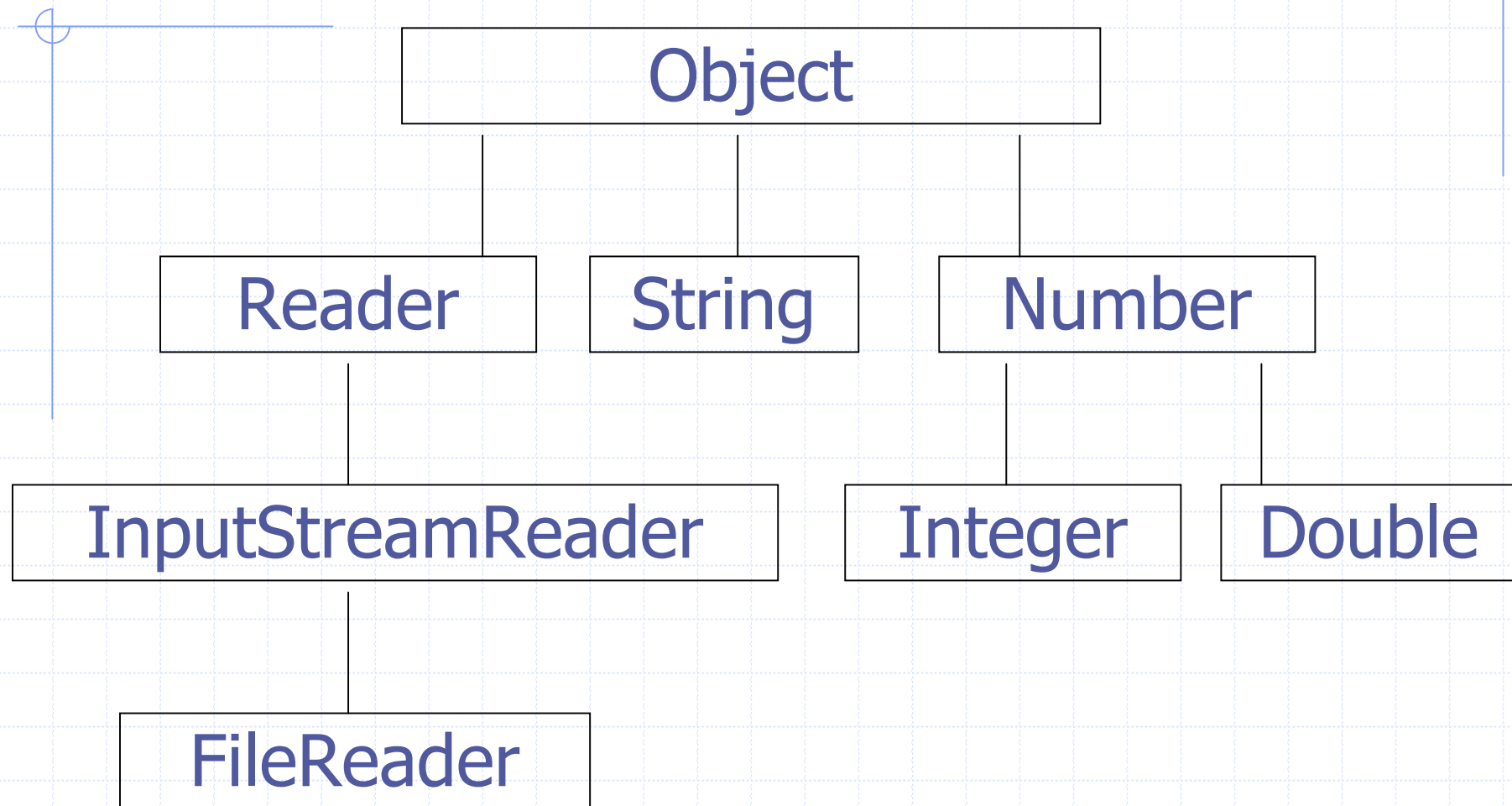
## ◆ Goals

- Robustness
- Adaptability
- Flexible code reuse

## ◆ Principles

- Abstraction
- Encapsulation
- Modularity

# Class Hierarchy



# Inheritance & Polymorphism

- ◆ subclass **extends** superclass
- ◆ the subclass “specialises”, and “inherits” from, the superclass
- ◆ field or method in subclass can **override** field or method in superclass
- ◆ the appropriate method is found by means of **dynamic dispatch**
- ◆ keywords: **this & super**

# Example: Progression class

```
public class Progression {  
    protected long first, cur;  
    Progression() { cur = first = 0; }  
    protected long firstValue() { cur = first; return cur; }  
    protected long nextValue() { return ++cur; }  
    public void printProgression( int n ) {  
        System.out.println( firstValue());  
        for( int i = 2; i <= n; i++ ) {  
            System.out.print( “ ” + nextValue()); }  
        System.out.println();  
    }  
}
```

# ArithProgression subclass

```
class ArithProgression extends Progression {  
    protected long inc; // increment  
    // parameter inc masks or overrides field inc  
    ArithProgression( long inc ) {this.inc = inc; }  
    ArithProgression() {this( 1 );} // calls other constructor  
    // this method overrides Progression.nextValue()  
    protected long nextValue() {  
        cur += inc;  
        return cur;  
    }  
    // inherits methods firstValue() and printProgression()  
}
```

# FibonacciProgression

```
class FibonacciProgression extends Progression {  
    long prev;  
    FibonacciProgression( long value1, long value2 ) {  
        first = value1; prev = value2 - value1; }  
    FibonacciProgression() { this( 0, 1 ); }  
    protected long nextValue() {  
        long temp = prev;  
        prev = cur;  
        cur += temp;  
        return cur;  
    }  
}
```

# Test Progression

```
class Tester {  
    public static void main( String[] args ) {  
        Progression prog;  
        prog = new ArithProgression();//default increment  
        prog.printProgression( 10 );  
        prog = new ArithProgression( 5 );// increment 5  
        prog.printProgression( 10 );  
        prog = new FibonacciProgression(); // default  
        prog.printProgression( 10 );  
        prog = new FibonacciProgression( 4, 6 );  
        prog.printProgression( 10 );  
    }  
}
```



# Test Progression

```
ArithProgression(); //default increment
```

```
0 1 2 3 4 5 6 7 8 9
```

```
ArithProgression( 5 ); // increment 5
```

```
0 5 10 15 20 25 30 35 40 45
```

```
FibonacciProgression(); // default
```

```
0 1 1 2 3 5 8 13 21 34
```

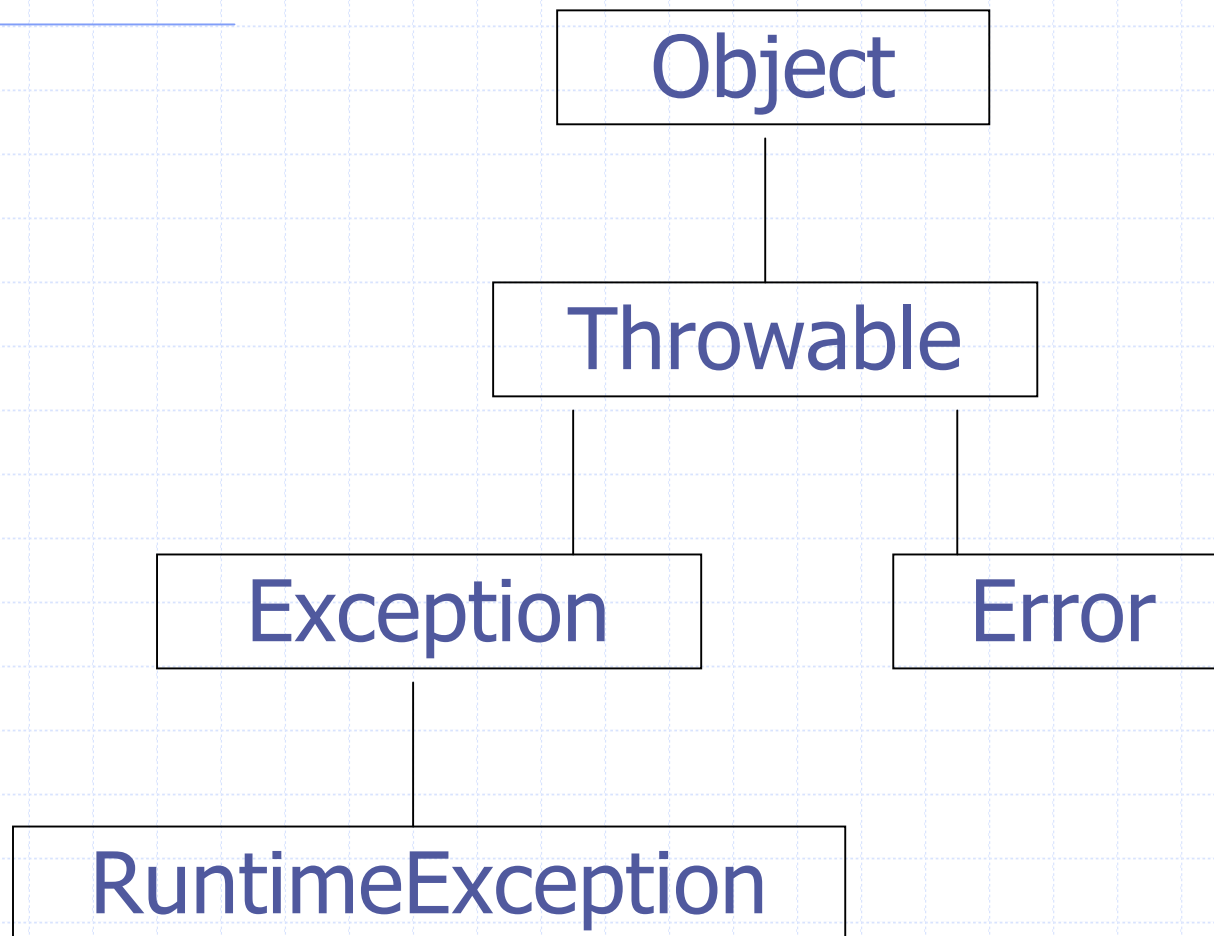
```
FibonacciProgression( 4, 6 );
```

```
4 6 10 16 26 42 68 110 178 288
```

# Exceptions

- ◆ when code encounters unexpected conditions, it **throws** an exception
- ◆ other code must then **catch** the exception
- ◆ when an exception is **thrown** in a method, the method may either **catch** the exception or **throw** it to the calling method
- ◆ Exceptions are in a hierarchy (and you can define your own)
- ◆ Exceptions other than **RuntimeExceptions** must be caught

# Exception Hierarchy



# Copy.java

```
try {  
    fileIn = new BufferedReader(new FileReader(source));  
    fileOut = new PrintWriter( new FileWriter( dest ));  
    String oneLine;  
    while(( oneLine = fileIn.readLine()) != null ) {  
        fileOut.println( oneLine ); }  
    } fileIn.close(); fileOut.close();  
catch( IOException e ) {  
    System.out.print( "Exception: " + e );  
    System.exit( 1 );  
}
```

# Interfaces

- ◆ An **interface** is a collection of method declarations with no data and no code
- ◆ When a class **implements** an interface, it must implement all methods declared in the interface
- ◆ Each class **extends** (directly) only one other class, but may **implement** more than one interface

# Abstract Classes

- ◆ an **abstract class** is a class containing empty method declarations
- ◆ abstract class cannot be **instantiated**
- ◆ subclass of an abstract class must provide **implementations** for all the abstract methods of the superclass

# Casting and instanceof

```
void printProgression( Progression prog ) {  
    .....  
    if( prog instanceof ArithProgression ) {  
        System.out.println( “int = ” +  
            ((ArithProgression) prog ).int );  
    }  
    if( prog instanceof FibonacciProgression ) {  
        System.out.println( “prev = ” +  
            ((FibonacciProgression) prog ).prev );  
    }  
}
```

# Javadoc

```
/**
```

```
 * Compute the sigmoid function of a given input
```

```
 *
```

```
 * @param x Any real number
```

```
 * @return y The sigmoid function of x
```

```
 */
```

```
Public double sigmoid( double x ) {  
    if( x >= 0.0 )  
        return 1.0 / ( 1.0 + Math.exp( -x ));  
    else  
        return 1.0 - sigmoid( -x );
```



# Design Patterns

- ◆ Position
- ◆ Iterator
- ◆ Template method
- ◆ Composition
- ◆ Comparator
- ◆ Locator
- ◆ Decorator