

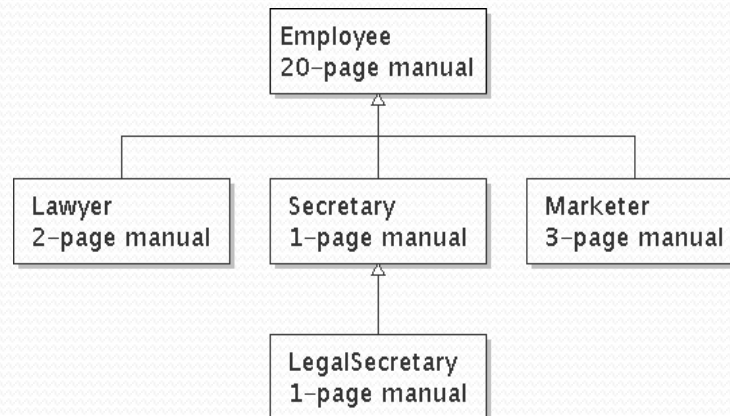
CSE 143

Lecture 25: Inheritance and Polymorphism



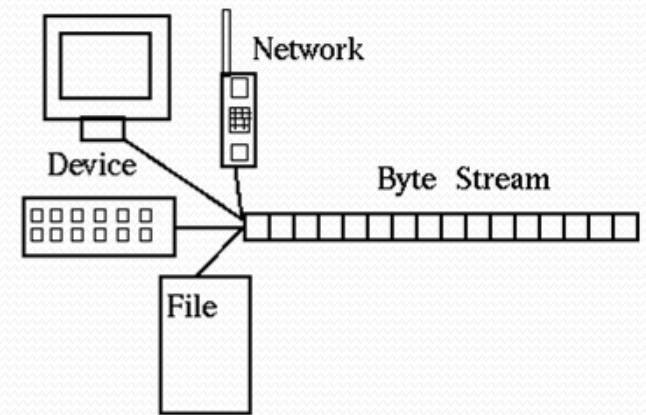
Recall: inheritance

- **inheritance**: Forming new classes based on existing ones.
 - a way to share/**reuse code** between two or more classes
 - **superclass**: Parent class being extended.
 - **subclass**: Child class that inherits behavior from superclass.
 - gets a copy of every field and method from superclass
 - **is-a relationship**: Each object of the subclass also "is a(n)" object of the superclass and can be treated as one.



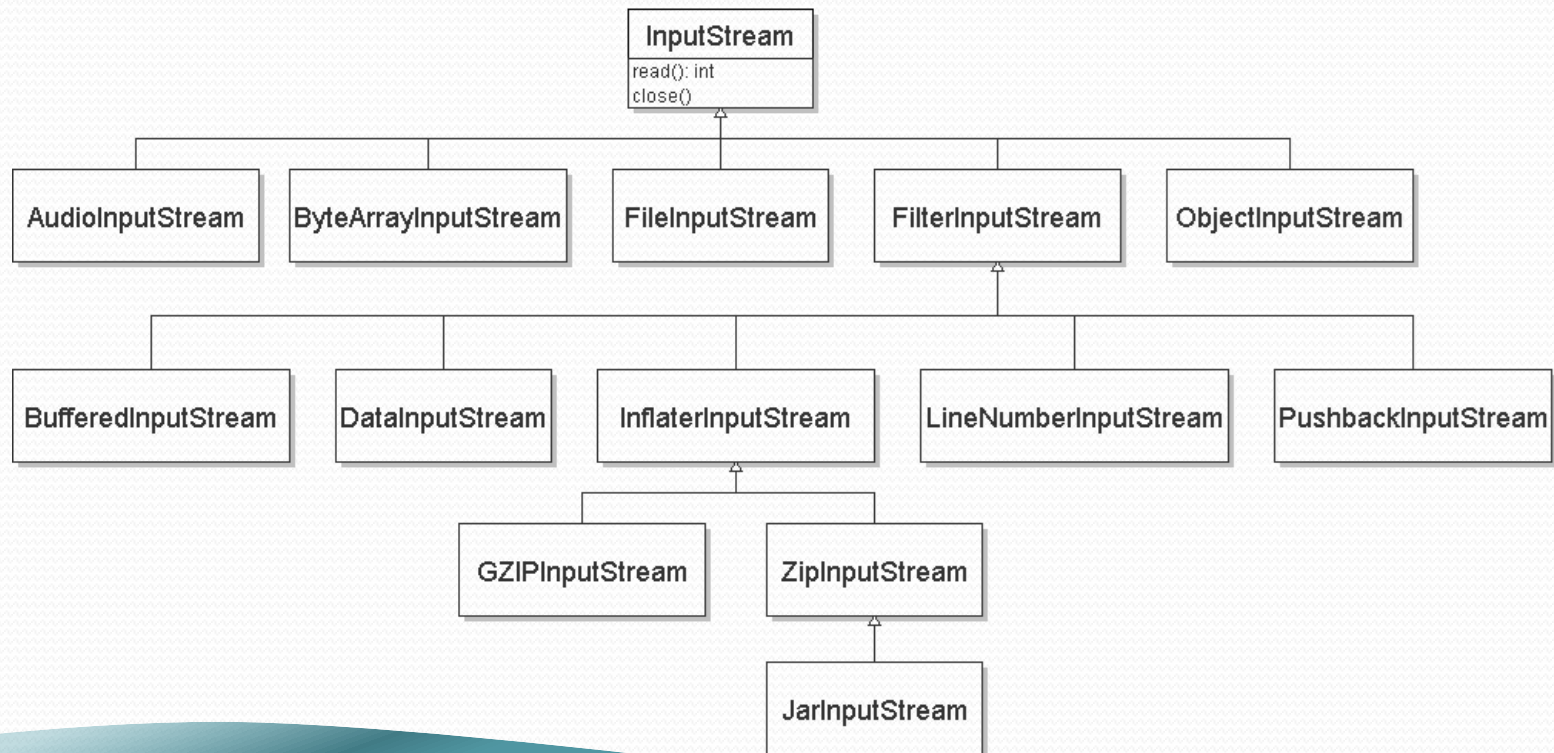
Input and output streams

- **stream**: an abstraction of a source or target of data
 - 8-bit bytes flow to (output) and from (input) streams
- can represent many data sources:
 - files on hard disk
 - another computer on network
 - web page
 - input device (keyboard, mouse, etc.)
- represented by `java.io` classes
 - `InputStream`
 - `OutputStream`



Streams and inheritance

- input streams extend common superclass `InputStream`; output streams extend common superclass `OutputStream`
 - guarantees that all sources of data have the same methods
 - provides minimal ability to read/write one byte at a time



Inheritance syntax

```
public class name extends superclass {  
  
public class Lawyer extends Employee {  
    ...  
}
```

- **override**: To replace a superclass's method by writing a new version of that method in a subclass.

```
public class Lawyer extends Employee {  
    // overrides getSalary method in Employee class;  
    // give Lawyers a $5K raise  
    public double getSalary() {  
        return 55000.00;  
    }  
}
```

super keyword

- Subclasses can call inherited behavior with `super`

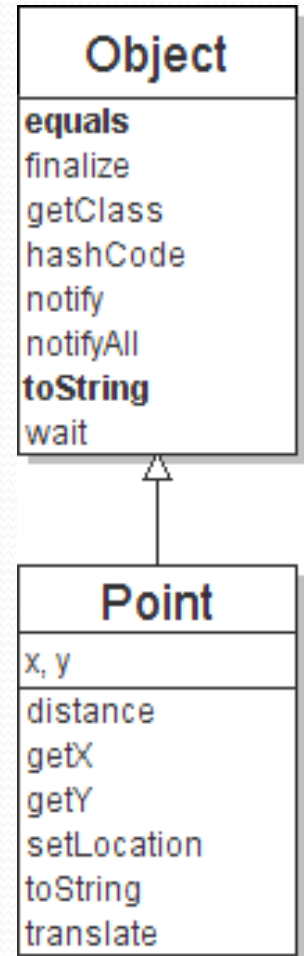
```
super.method (parameters)  
super (parameters) ;
```

```
public class Lawyer extends Employee {  
    public Lawyer(int years) {  
        super(years); // calls Employee constructor  
    }  
  
    // give Lawyers a $5K raise  
    public double getSalary() {  
        double baseSalary = super.getSalary();  
        return baseSalary + 5000.00;  
    }  
}
```

- Lawyers now always make \$5K more than Employees.

The class Object

- The class `Object` forms the root of the overall inheritance tree of all Java classes.
 - Every class is implicitly a subclass of `Object`
- The `Object` class defines several methods that become part of every class you write. For example:
 - `public String toString()`
Returns a text representation of the object, usually so that it can be printed.



Object methods

method	description
<code>protected Object clone()</code>	creates a copy of the object
<code>public boolean equals(Object o)</code>	returns whether two objects have the same state
<code>protected void finalize()</code>	used for garbage collection
<code>public Class<?> getClass()</code>	info about the object's type
<code>public int hashCode()</code>	a code suitable for putting this object into a hash collection
<code>public String toString()</code>	text representation of object
<code>public void notify()</code> <code>public void notifyAll()</code> <code>public void wait()</code> <code>public void wait(...)</code>	methods related to concurrency and locking (take a data structures course!)

Using the Object class

- You can store any object in a variable of type `Object`.

```
Object o1 = new Point(5, -3);  
Object o2 = "hello there";
```

- You can write methods that accept an `Object` parameter.

```
public void checkNotNull(Object o) {  
    if (o != null) {  
        throw new IllegalArgumentException();  
    }  
}
```

- You can make arrays or collections of `Objects`.

```
Object[] a = new Object[5];  
a[0] = "hello";  
a[1] = new Random();  
List<Object> list = new ArrayList<Object>();
```

Polymorphism

- **polymorphism:** Ability for the same code to be used with different types of objects and behave differently with each.
- A variable or parameter of type T can refer to any subclass of T .

```
Employee ed = new Lawyer();  
Object otto = new Secretary();
```

- When a method is called on `ed`, it behaves as a `Lawyer`.
- You can call any `Employee` methods on `ed`.
You can call any `Object` methods on `otto`.
 - You can *not* call any `Lawyer`-only methods on `ed` (e.g. `sue`).
You can *not* call any `Employee` methods on `otto` (e.g. `getHours`).

Polymorphism examples

- You can use the object's extra functionality by casting.

```
Employee ed = new Lawyer();  
ed.getVacationDays();           // ok  
ed.sue();                       // compiler error  
(Lawyer ed).sue();             // ok
```

- You can't cast an object into something that it is not.

```
Object otto = new Secretary();  
System.out.println(otto.toString()); // ok  
otto.getVacationDays();             // compiler error  
(Employee otto).getVacationDays(); // ok  
(Lawyer otto).sue();               // runtime error
```

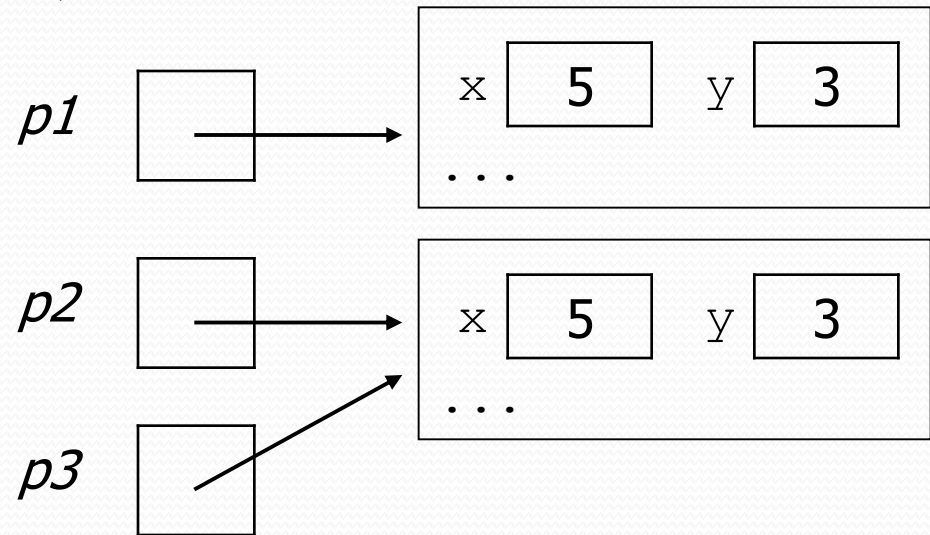
Recall: comparing objects

- The `==` operator does not work well with objects.
 - It compares references, not objects' state.
 - It produces `true` only when you compare an object to itself.

```
Point p1 = new Point(5, 3);  
Point p2 = new Point(5, 3);  
Point p3 = p2;
```

```
// p1 == p2 is false;  
// p1 == p3 is false;  
// p2 == p3 is true
```

```
// p1.equals(p2)?  
// p2.equals(p3)?
```



Default equals method

- The Object class's equals implementation is very simple:

```
public class Object {  
    ...  
    public boolean equals(Object o) {  
        return this == o;  
    }  
}
```

- However:
 - When we have used equals with various objects, it didn't behave like == . Why not? `if (str1.equals(str2)) { ...`
 - The [Java API documentation for equals](#) is elaborate. Why?

Implementing equals

```
public boolean equals(Object name) {  
    statement(s) that return a boolean value ;  
}
```

- The parameter to `equals` must be of type `Object`.
- Having an `Object` parameter means *any* object can be passed.
 - If we don't know what type it is, how can we compare it?

The instanceof keyword

```
if (variable instanceof type) {  
    statement(s);  
}
```

- Asks if a variable refers to an object of a given type.
 - Used as a boolean test.

```
String s = "hello";  
Point p = new Point();
```

expression	result
s instanceof Point	false
s instanceof String	true
p instanceof Point	true
p instanceof String	false
p instanceof Object	true
s instanceof Object	true
null instanceof String	false
null instanceof Object	false

equals method for Points

```
// Returns whether o refers to a Point object with
// the same (x, y) coordinates as this Point.
public boolean equals(Object o) {
    if (o instanceof Point) {
        // o is a Point; cast and compare it
        Point other = (Point) o;
        return x == other.x && y == other.y;
    } else {
        // o is not a Point; cannot be equal
        return false;
    }
}
```


More about equals

- Equality is expected to be reflexive, symmetric, and transitive:

```
a.equals(a) is true for every object a
a.equals(b) ↔ b.equals(a)
(a.equals(b) && b.equals(c)) ↔ a.equals(c)
```

- No non-null object is equal to null:

```
a.equals(null) is false for every object a
```

- Two sets are equal if they contain the same elements:

```
Set<String> set1 = new HashSet<String>();
Set<String> set2 = new TreeSet<String>();
for (String s : "hi how are you".split(" ")) {
    set1.add(s);    set2.add(s);
}
System.out.println(set1.equals(set2));    // true
```

I/O and exceptions

- **exception**: An object representing an error.
 - **checked exception**: One that must be handled for the program to compile.
- Many I/O tasks throw exceptions.
 - Why?
- When you perform I/O, you must either:
 - also **throw** that exception yourself
 - **catch** (handle) the exception



Throwing an exception

```
public type name (params) throws type {
```

- **throws clause:** Keywords on a method's header that state that it may generate an exception.

- Example:

```
public void processFile(String filename)  
    throws FileNotFoundException {
```

"I hereby announce that this method might throw an exception, and I accept the consequences if it happens."

Catching an exception

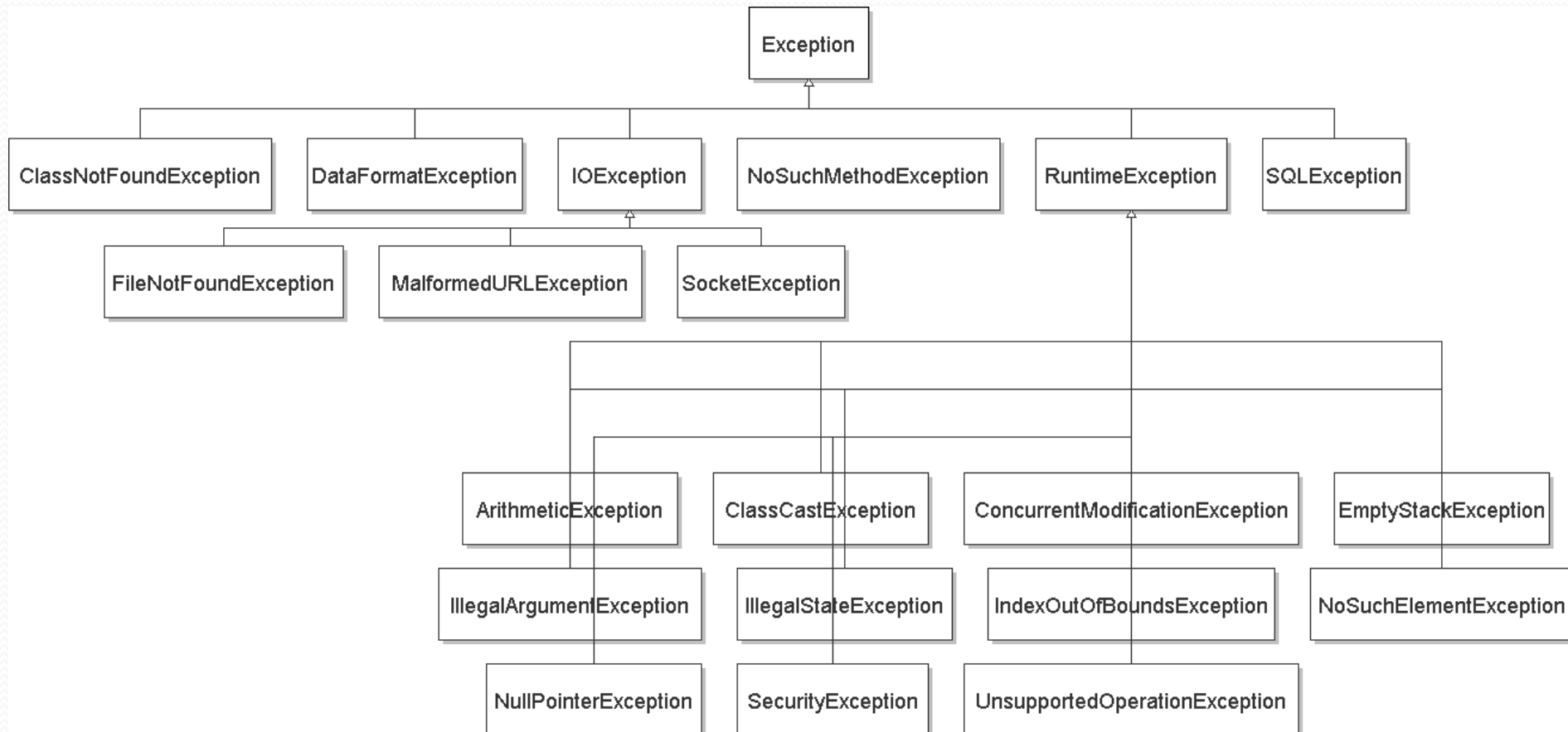
```
try {  
    statement(s);  
} catch (type name) {  
    code to handle the exception  
}
```

- The `try` code executes. If the given exception occurs, the `try` block stops running; it jumps to the `catch` block and runs that.

```
try {  
    Scanner in = new Scanner(new File(filename));  
    System.out.println(input.nextLine());  
} catch (FileNotFoundException e) {  
    System.out.println("File was not found.");  
}
```

Exception inheritance

- Exceptions extend from a common superclass `Exception`



Dealing with an exception

- All exception objects have these methods:

Method	Description
<code>public String getMessage ()</code>	text describing the error
<code>public String toString ()</code>	a stack trace of the line numbers where error occurred
<code>getCause (), getStackTrace (), printStackTrace ()</code>	other methods

- Some reasonable ways to handle an exception:
 - try again; re-prompt user; print a nice error message; quit the program; do nothing (!)

Inheritance and exceptions

- You can catch a general exception to handle any subclass:

```
try {
    Scanner input = new Scanner(new File("foo"));
    System.out.println(input.nextLine());
} catch (Exception e) {
    System.out.println("File was not found.");
}
```

- Similarly, you can state that a method throws any exception:

```
public void foo() throws Exception { ...
```

- Are there any disadvantages of doing so?