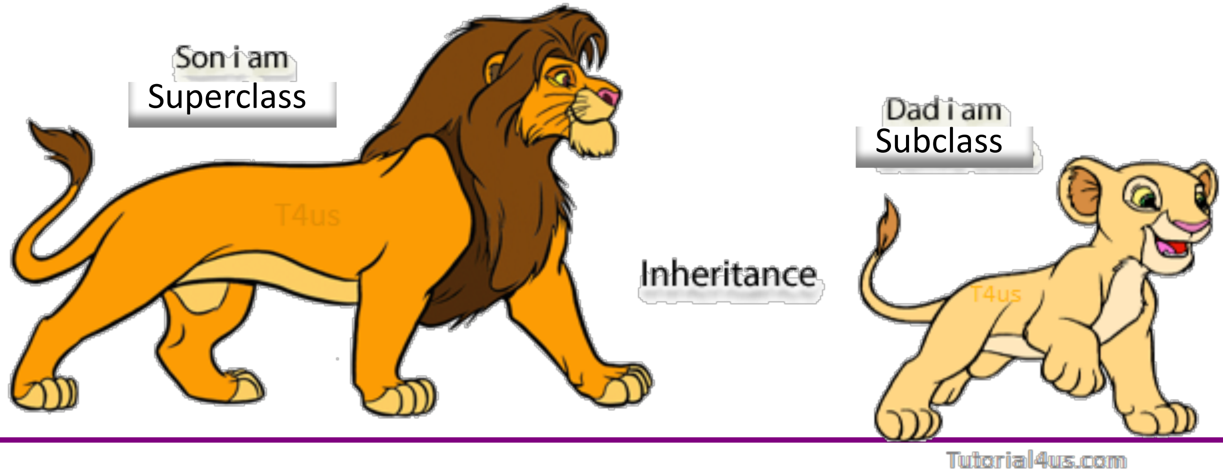
RECAP of Lectures 1&2

Maria Zontak

‘Is-a’ in Programming

Java, C++ and more provide direct support for “IS - A”:

- **Class Inheritance -** new class **extends** existing class
- Key for good object-oriented programming:
  - Using the SAME code in MANY contexts → Reusable code
  - Reduce bugs → Robust and maintainable
- **Terminology:**

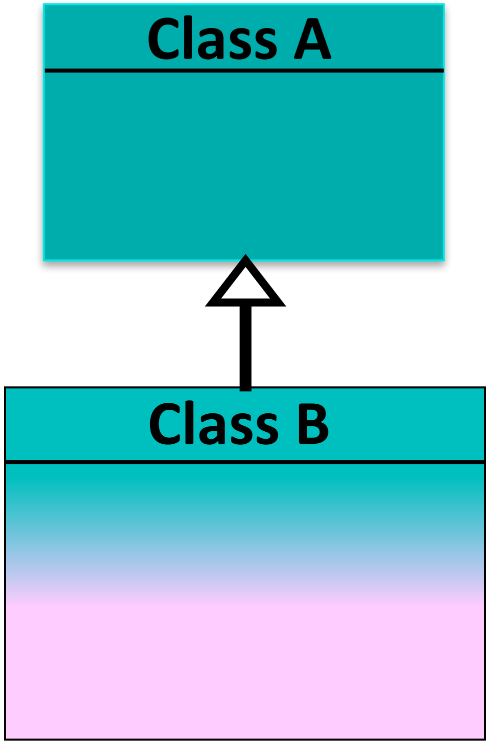


Vocabulary and Principles

Original/Extend**ed** class - called **base class** or **super class**

New/Extend**ing** class - called **derived class** or **sub class**

UML for B extends A



Derived class

- automatically *inherits* from the base class all public/protected instance variables and methods
- can *add* additional methods and instance variables
- can provide *different versions* of inherited methods → **override**

Member Access

public:

- accessible anywhere the class can be accessed

private:

- accessible only inside the same class
- Does *not* include subclasses – derived classes have no special permissions

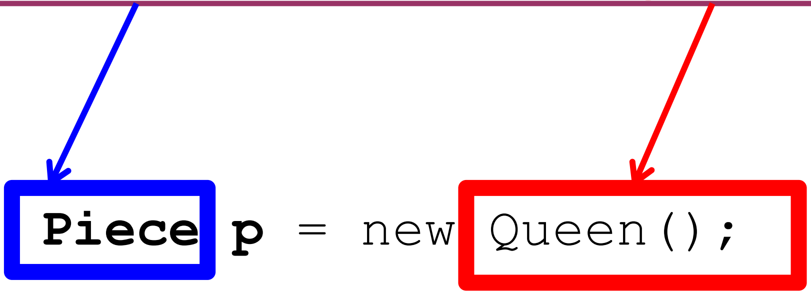
A new mode: ***protected***

- accessible inside the defining class and all its subclasses

When to use public/private/protected:

- NEVER use public for fields
- Use protected for "internal" things that subclasses also are intended to access

Static and Dynamic Types



```
Piece p = new Queen();
```

- Static/compile time type : the declared type of the reference variable. Used by the compiler to check syntax.

- Dynamic/runtime-time type: the object type the variable currently refers to (can change as program executes)

- Interface and Abstract Classes define a TYPE , which one?

Dynamic Dispatch→ Where to look for methods?

```
Piece p = new Queen();
```

- **Static** types - the compiler knows exactly what method must execute
- **Dynamic** types - the compiler knows the *name* of the method but…

There could be ambiguity about which version of the method will actually be needed at run-time:

- The decision is deferred until run-time → dynamic dispatch
- The chosen method matches the dynamic (actual) type of the object

Public Interface

WHAT?

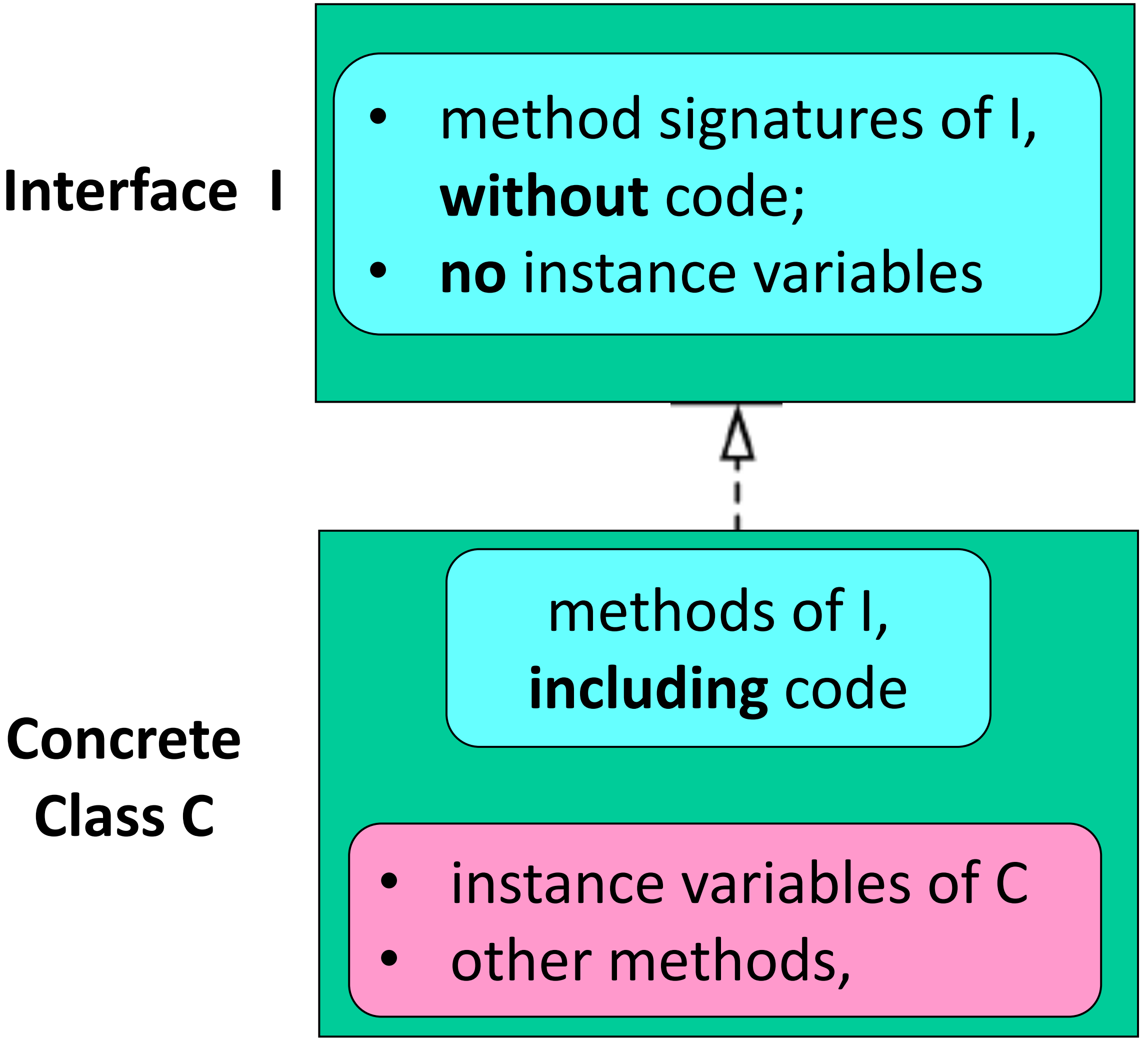
- **A set of method declarations/common behaviors**
- **Contract** /protocol of what the classes can do.

→Class that agrees to interface, should implement its behaviors

WHY needed?

- Allows interaction, without knowing specific implementation
- Take advantage of **multiple inheritance for one class**.
- **Achieves subtype polymorphism →**

Classes that implement the same interface can be treated similarly



Interface I

- method signatures of I, **without** code;
- **no** instance variables

Concrete Class C

methods of I, **including** code

- instance variables of C
- other methods,

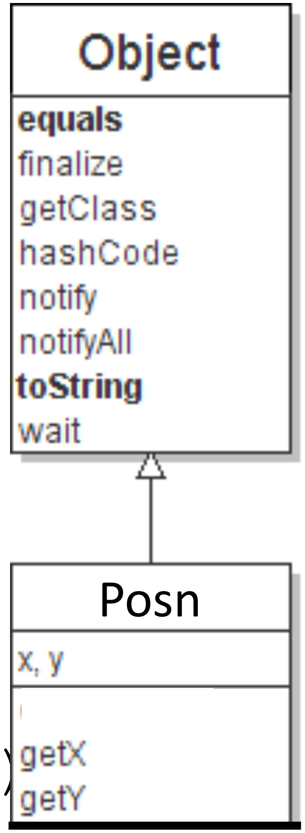
Interfaces vs Inheritance

	INTERFACE	INHERITANCE
"Is –A" Relationship	🟩	🟩
Code Sharing	🟩	🟩
B→A	B *implements* interface A → B inherits the method signatures from A (must implement them) **Specification**	B *extends* class A → B inherits everything from A (including any method code and instance variables) **Implementation**

Class `Object`

- All types of objects have a superclass named `Object`.
  - Every class implicitly extends `Object`

- The `Object` class defines several methods:
  - `public String toString()`
    Returns a text representation of the object, often so that it can be printed.
  - `public boolean equals(Object other)`
    Compare the object to any other for equality. Returns `true` if the objects have equal state.



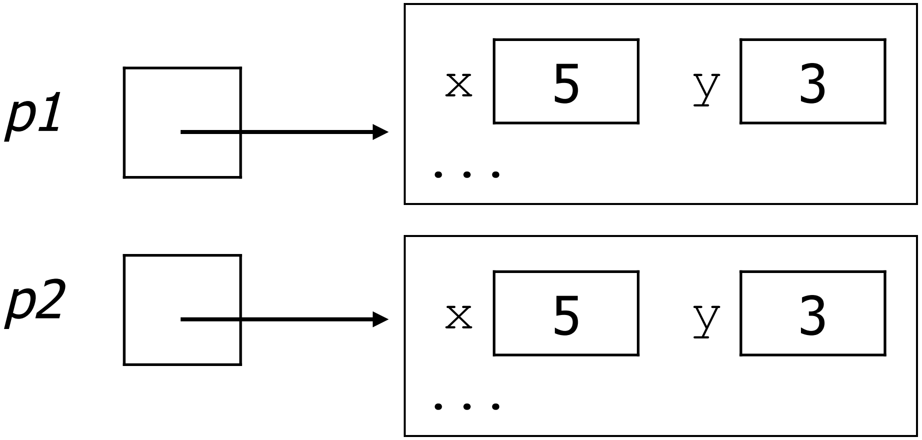
Recall: comparing objects

- The == operator does not work well with objects.

== compares references to objects, not their state.

It only produces `true` when you compare an object to itself.

```
Posn p1 = new Posn(5, 3);
Posn p2 = new Posn(5, 3);
if (p1 == p2) {
    System.out.println("equal");
}
```



Flawed `equals` method

We can change this behavior by writing an `equals` method.

- Ours will *override* the default behavior from class `Object`.
- The method should compare the state of the two objects and return `true` if they have the same x/y position.

A flawed implementation:

```
public boolean equals(Posn o) {
  if (this.x != null ? !this.x.equals(o.x) : o.x != null)
                                                   return false;
  return this.y != null ? this.y.equals(o.y) : o.y == null;
}
```

`equals` and `Object`

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

- The parameter to `equals` must be of type `Object`.
- `Object` is a general type that can match any object.
- Having an `Object` parameter means *any* object can be passed.

If we do not know what type it is, how can we compare it?

Another flawed version

- Another flawed `equals` implementation:

```
public boolean equals(Object o) {
 if (this.x != null ? !this.x.equals(o.x) : o.x !=null)
                                                   return false;
 return this.y != null ? this.y.equals(o.y) : o.y == null;
 }
```

- It does not compile:

```
Posn.java:36: cannot find symbol
symbol  : variable x
…
```

- The compiler is saying,
  "`o` could be any object. Not every object has an `x` field."

Type-casting objects

- Solution: *Type-cast* the object parameter to a Posn.

```
public boolean equals(Object o) {

  Posn posn = (Posn) o;

 if (this.x != null ? !this.x.equals(posn.x) : posn.x !=null)
                                                return false;
  return this.y != null ? this.y.equals(posn.y) : posn.y == n
  }
```

- Casting objects is different than casting primitives.
  - Really casting an Object reference into a Posn reference.
  - Does NOT actually change the object that was passed.
  - Tells the compiler to *assume* that o refers to a Posn object.

Comparing different types

```
Posn p = new Posn(7, 2);
if (p.equals("hello")) {    // should be
false
     ...
}
```

- Currently our method crashes on the above code:

```
Exception in thread "main"
java.lang.ClassCastException: java.lang.String
        at Posn.equals(Posn.java:25)
```

- The culprit is the line with the type-cast:

```
public boolean equals(Object o) {
    Posn posn = (Posn) o;
```

What about this?

```
public boolean equals(Object o) {



  if (this.getClass()!=o.getClass()) return false;
  Posn posn = (Posn) o;
  if (this.x != null ? !this.x.equals(posn.x) : posn.x !=null)
                                          return false;
  return this.y != null ? this.y.equals(posn.y) : posn.y == n
}
```

What about this?

```
public boolean equals(Object o) {


  if (o==null || this.getClass()!=o.getClass()) return false;
   Posn posn = (Posn) o;
   if (this.x != null ? !this.x.equals(posn.x) : posn.x !=null)
                                                return false;
   return this.y != null ? this.y.equals(posn.y) : posn.y == n
 }
```

Finally

```
public boolean equals(Object o) {

  if (this == o) return true;
  if (o==null || this.getClass()!=o.getClass()) return false;
  Posn posn = (Posn) o;
  if (this.x != null ? !this.x.equals(posn.x) : posn.x !=null)
                                     return false;
  return this.y != null ? this.y.equals(posn.y) : posn.y == n
}
```