Implements vs. Extends When Defining a Class

- **implements:**
  - Keyword followed by the name of an INTERFACE
  - Interfaces only have method PROTOTYPES
  - You **CANNOT** create an object of an interface type

- **extends:**
  - Keyword followed by the name of a BASE CLASS
  - Base class contains method IMPLEMENTATIONS
  - Allows INHERITANCE!!
  - You **CAN** create objects of that base class type
Example: People at University

- Base class: person
- Derived classes: student, faculty, administrator
- Derived from those: undergrad, grad, instructor, professor,…

```
Person
    \--- Student
        \----- Undergrad
        \----- GradStudent
    \--- Faculty
        \----- Instructor
        \----- Professor
    \--- Administrator
        \----- ...
```

University Person Example

```java
class: Person
instance variables:
String name
String idNum
methods:
Person( ... ) [various]
String getName( )
String getIdNum( )
void setName( String )
void setIdNum( String )
String toString( )
boolean equals( Person )

class: Student
instance variables:
int admitYear
double gpa
methods:
Student( ... ) [various]int getAdmitYear( )
double getGpa( )
void setAdmitYear( int )
void setGpa( double )
String toString( )
boolean equals( Student )

class: Faculty
instance variables:
int hireYear
methods:
Faculty( ... ) [various]int hireYear( )
void setHireYear( int )
String toString( int )
boolean equals( Faculty )

extends Person

extends Person
```
Person class...

Lets inspect the (simple) code.
Student class?

Memory diagram?

“this” vs. “super”

Consider:
  Person q = new Student();
Implementing a subclass

Lets implement the Student class…

Constructors:
- Typical (with parameters)
- No arg Constructor
- Copy Constructor

Overridden methods:
- toString
- equals
Overriding vs. Overloading

- **Overriding**: a derived class defines a method with same name, parameters as base class
- **Overloading**: two or more methods have the same name, but different parameters
- **Example**

```java
public class Person {
    public void setName(String n) { name = n; }
    ...
}

public class Faculty extends Person {
    public void setName(String n) {
        super.setName("The Evil Professor " + n);
    }
    public void setName(String first, String last) {
        super.setName(first + " " + last);
    }
}
```

- **Base class setName( )**
- **Override**
- **Overloading**
Early vs. Late Binding

Consider:

```java
Person p = new Student();
System.out.println( p.toString() );
```

Which version of `toString`—Person or Student—is called?

- **Early (static) binding**
  - `p` is declared to be of type `Person`
  - Therefore, the `Person` version of `toString` is used

- **Late (dynamic) binding**
  - The object to which `p` refers was created as `Student` object
  - Therefore, the `Student` version of `toString` is used

Java uses late binding (C++ by default uses early binding)

- Early binding is more runtime efficient (decisions about method versions can be made at compile time)
- Late binding respects encapsulation (object defines its operations when it is created)
Polymorphism

- Java’s **late binding** makes it possible for a single reference variable to refer to objects of many different types. Such a variable is said to be **polymorphic** (meaning having many forms).
- **Example**: Create an array of various university people and print.

```java
Person[ ] list = new Person[3];
list[0] = new Person("Col. Mustard", "000-00-0000");
list[1] = new Student("Ms. Scarlet", "111-11-1111", 1998, 3.2);
list[2] = new Faculty("Prof. Plum", "222-22-2222", 1981);
for ( int i = 0; i < list.length; i++ )
   System.out.println( list[i].toString( ) )
```

**Output:**

```
[Col. Mustard] 000-00-0000
[Ms. Scarlet] 111-11-1111 1998 3.2
[Prof. Plum] 222-22-2222 1981
```

- **What type is list[i] meanings?** It can be a reference to any object that is derived from Person. The appropriate toString will be called.
Public, Protected, Package(default) and Private

- Select which level of visibility

<table>
<thead>
<tr>
<th>Access Level/Group</th>
<th>Access Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Class</td>
</tr>
<tr>
<td>public</td>
<td>Y</td>
</tr>
<tr>
<td>protected (avoid)</td>
<td>Y</td>
</tr>
<tr>
<td>package (default)</td>
<td>Y</td>
</tr>
<tr>
<td>private</td>
<td>Y</td>
</tr>
</tbody>
</table>
Shadowing

- Can we override instance variables just like methods?
- Yes, but be careful!
  - Overriding instance variable is called **shadowing**
  - Shadowing hides instance variables of base class (can still access them using `super.varName` in subclass, but not in “outside world”)

```java
public class Person {  
    String name;
    ...
}

public class Administrator extends Person {  
    String name;  // name refers to Administrator’s name
}
```

- Confusing! Better to pick a new variable name
Example of Overloading/Overriding

public class Base {
    public void m (int x) { ... }
}

public class Derived extends Base {
    public void m (int x) { ... }
    public int m (int x) { ... }
    public void m (double d) { ... }
}

// The following appears in the same package as above

Base b = new Base( );
Base d = new Derived( );
Derived e = new Derived( );
    b.m (5);
    d.m (6);
    d.m (7.0);
    e.m (8.0);

Error! Since d is declared Base, the compiler looks for Base:m(double)
Doesn't exist! So this does not make it past the compiler, even
though Derived:m(double) is defined!
Object

- Recall: inheritance induces “is-a” hierarchy on classes
  - Undergrad “is-a” Student
  - Student “is-a” Person
  - etc.
- Person “is-a” ....?
- Person “is-a”(n) Object
- Student “is-a”(n) Object

```
Object
  ↓
Person
  ↓
  Student
  ↓
  Undergrad
  ↓
  GradStudent
  ↓
  Faculty
  ↓
  Instructor
  ↓
  Professor
  ↓
  Administrator
  ...
  ...
```
More on Object

- Special class at top of class inheritance hierarchy
- Defined in `java.lang`
- Every class is derived (either directly or indirectly) from `Object`
  - e.g.
    ```java
    public class Foo { ...}
    ```
    is equivalent to
    ```java
    public class Foo extends Object {...}
    ```

- Structure of Object
  - No instance variables
  - A number of methods, including:
    - `toString()`
    - `equals(Object o)`

Let’s look at the Javadoc...
Class vs. Type Information

- In Java
  - Every object is in one class (the one it was created from using `new`)
  - Objects may have many types (all those that class is based on)
    - Interfaces
    - Superclasses

- E.g. consider
  
  ```java
  Student bob = new Student();
  Person p = bob;
  
  Class of object pointed to by `bob` and `p` is `Student`
  
  Type of object can be `Student`, `Person`, `Object`, etc.
  ```
Accessing Type Information

- `instanceof`
  - Java boolean operator (not a method)
  - Returns true if given object “is-a”(n) object of given type
  - E.g.

    ```java
    Student carol = new Student ( ... );
    if (carol instanceof Person) // true, because carol “is-a” Person
    ```
Object Casting

- Recall casting in primitive types
  - Casting: conversion of elements from one type to another
  - Widening Conversion (always OK)
    - `double x = 3;` // 3 (int) widening conversion to double
  - Narrowing Conversion
    - Must use explicit type conversions to perform this casting
      - `int x = (int)3.0;` // 3.0 explicitly cast to int

- Similar notions can be found with object types also
  - Upcasting
    - Casting a reference to a superclass (casting up the inheritance tree)
    - Always OK
    - Just ignore the parts that were added by the subclass
  - Downcasting
    - Casting a reference to a derived class
    - Requires explicit casting operator
    - Can cause runtime error
### Object Casting

Person \( p = \text{new} \) Person();
Student \( s = \text{new} \) Student();
Person tricky = new Student();

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Student} \ y = p; )</td>
<td>Downcast – Does not compile</td>
</tr>
<tr>
<td>( \text{Student} \ y = (\text{Student})p; )</td>
<td>Compiles, but throws exception</td>
</tr>
<tr>
<td>( \text{Student} \ y = \text{tricky}; )</td>
<td>Downcast – Does not compile</td>
</tr>
<tr>
<td>( \text{Student} \ y = (\text{Student})\text{tricky}; )</td>
<td>Works fine</td>
</tr>
<tr>
<td>( \text{(Faculty)}s )</td>
<td>Does not compile</td>
</tr>
<tr>
<td>( \text{(Faculty)}\text{tricky} )</td>
<td>Compiles, but throws Exception</td>
</tr>
</tbody>
</table>