Inheritance

Theory

- Inheritance is needed when multiple classes are very similar in nature
- The inheritance relationship is called the "**is-a(n)**" relationship
- Thankfully this means we save some coding since it will be shared

Below are two example classes:

	Employees	
Full Time Professor		**Part Time Professor**
last name		**last name**
first name		**first name**
Ssn		**ssn**
Address		**address**
Birthdate		**birthdate**
vacation days		**contact number**
health insurer		**Employer**
office phone		
Tenor		
sick days		

- Notice how both classes are very similar in many aspects.
- Also, we can create other classes (like Secretary, Janitor, etc…) that would again use many of the same variables, but may have different members pertaining to their job responsibility.
- Also, either a Full Time Professor or Part Time Professor "is – a(n)" Employee.
- Some other examples of the "is a(n)" inheritance relationship
    - A student is a person
    - A professor is a faculty member
    - A lecturer is a faculty member

    - An adjunct professor is a faculty member

Breaking it down - The base (super) class

- Since so many classes shared many variables, lets create a BASE CLASS, that will let others called **sub or derived** classes share the common variables and methods/functions
- also called a **super or base** class, since the keyword super will be used to access the base classes' methods

	BASE (SUPER) CLASS	
	Employee	
SUB (DERIVED) CLASS		SUB CLASS
Full Time Professor	last name	**Part Time Professor**
	first name	
vacation days	ssn	contact number
health insurer	birthdate	Employer
office phone	address	
tenor		
sick days		

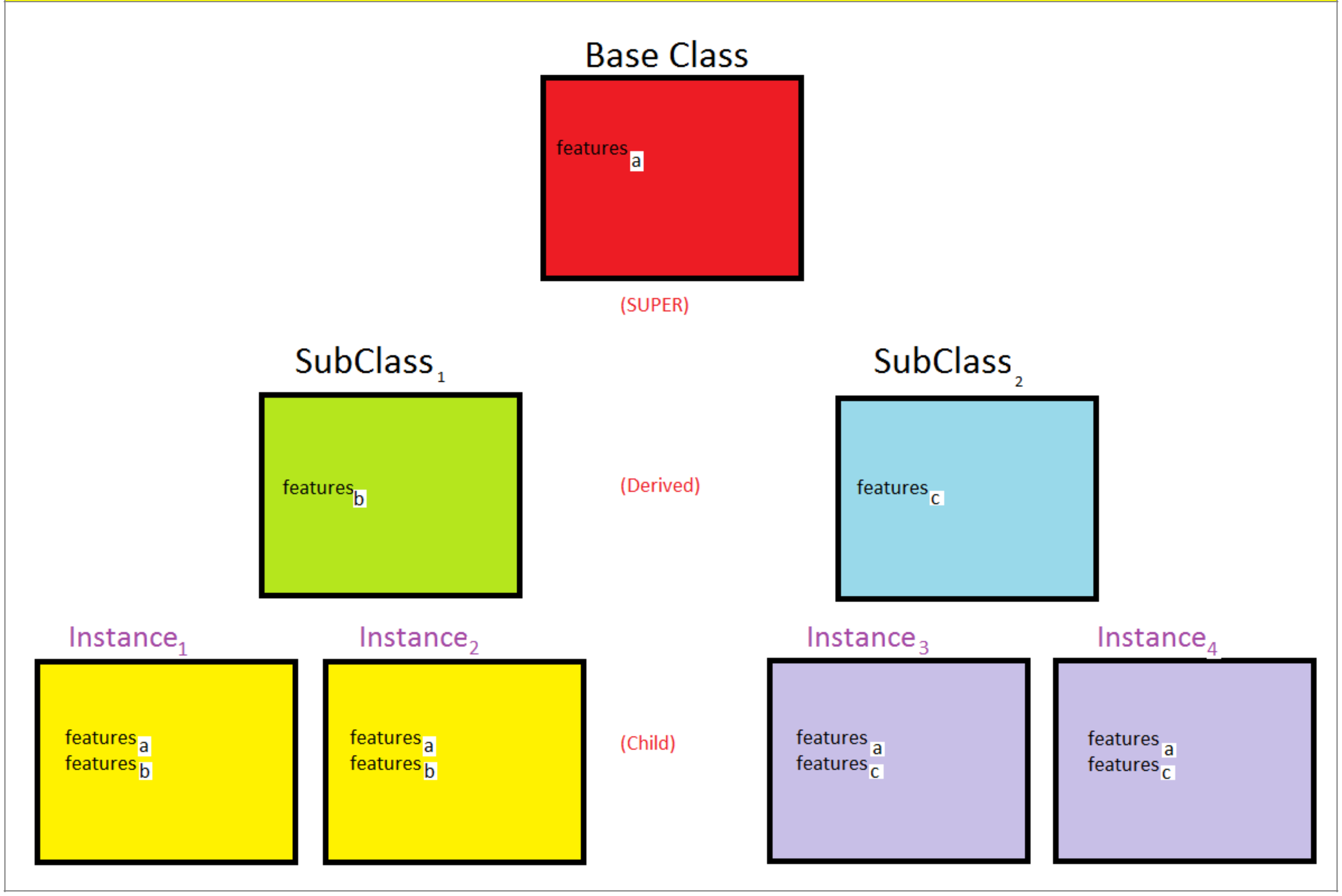
In Practice

- we are creating more specialized subclass (objects)
  - a specific task that adds to the super (base class)
- all objects should (at minimum) have a complete class profile
  - accessors
  - mutators
  - etc…
- with this established, we can reuse for other projects we might get later
- subclasses may be
  - much smaller than the base class
  - have some of their own member variables/methods/"behaviors"

Visual representation of Base & Sub Classes

- a visual understanding may be of some help
- please notice
  - an instance of a subclass will inherit both features of the subclass AND base
  - how the distinct players in this process are also called (super, derived, …)

Visually understanding Inheritance



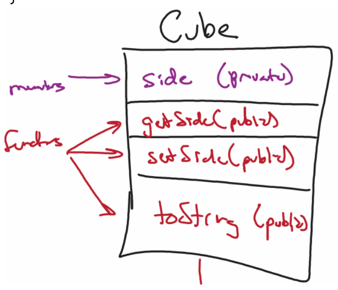
The Extends Reserved word

- extends can be thought of as a “better version”, more detailed version
  - you are “extending a template”
- used only in the subclass
- notice the naming convention **for objects** below
  - base class has a simple name
  - extended class (subclass) has the Extended and Base classes object
- “extends” induces a **subtyping** relation
  - Part Time professor would be a “subtype” of Employee
- S <: T
  - (S is a subtype of T)
- P ⊢ Q   (called a turnstile)
  - means that Q is derivable from P
  - Or from P, I know that Q…

Overall Example with Extends

Cube and Extended Cube Example

```
public class Cube
{
      private double side;

      public void setSide(double x) {side = x; }
      public double getSide() { return side; }
      public String toString() { return "This cube's sides are" + getSide(); }
}
```



```
public class ExtendedCube extends Cube
{
      public double getSurfaceArea() { return 6 * getSide() * getSide(); }
      public double getVolume() { return getSide() * getSide() *
getSide(); }
      public String toString()
      {
            return "This cube's sides is" + getSide() + "\n" +
            "This cube's surface area is" + getSurfaceArea() + "\n" +
            "This cube's sides are" + getVolume() + "\n";
      }
}
```

```
public class Driver
{
      public static void main(String args[])
      {
            Cube basic = new Cube();
            ExtendedCube advanced = new ExtendedCube();

            basic.setSide(6);
            advanced.setSide(10);

            System.out.println(basic);
            System.out.println(advanced);
      }
}
```

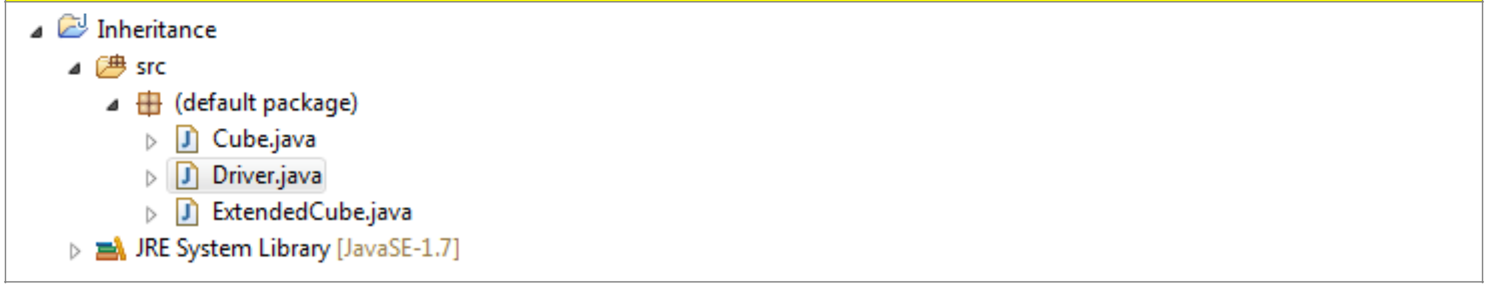
1. The "advanced" instance above called what method that IT DIDN'T have. Which one was it?? Why was it able to call that method??

2. Draw what the ExtendedCube would look like (show relation to Cube (arrow))
   Answer$_b$:
3. Draw what the INSTANCES look like, and what they should return as values
4. Copy and Paste the code into your IDE and run.

Inheritance and the file/project makeup

- This example uses a default package
  - So really no package
  - Notice that all of the .java files are together

Inheritance File Setup



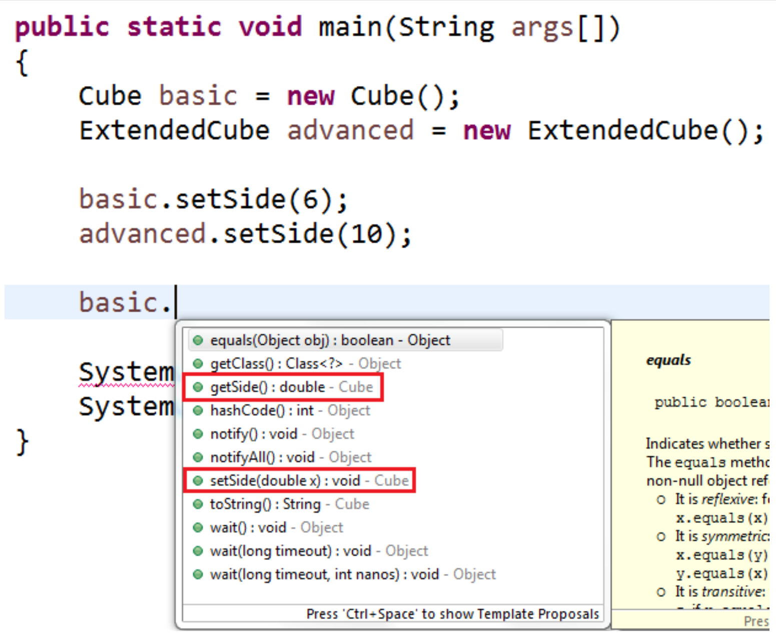
```
Inheritance
  src
    (default package)
      Cube.java
      Driver.java
      ExtendedCube.java
  JRE System Library [JavaSE-1.7]
```

Eclipse can help with Inheritance

- Again, let the IDE help you
- As long as you have BOTH the base and derived classes setup, the IDE will tell you what options you have

IDE help with Inheritance

Base Class

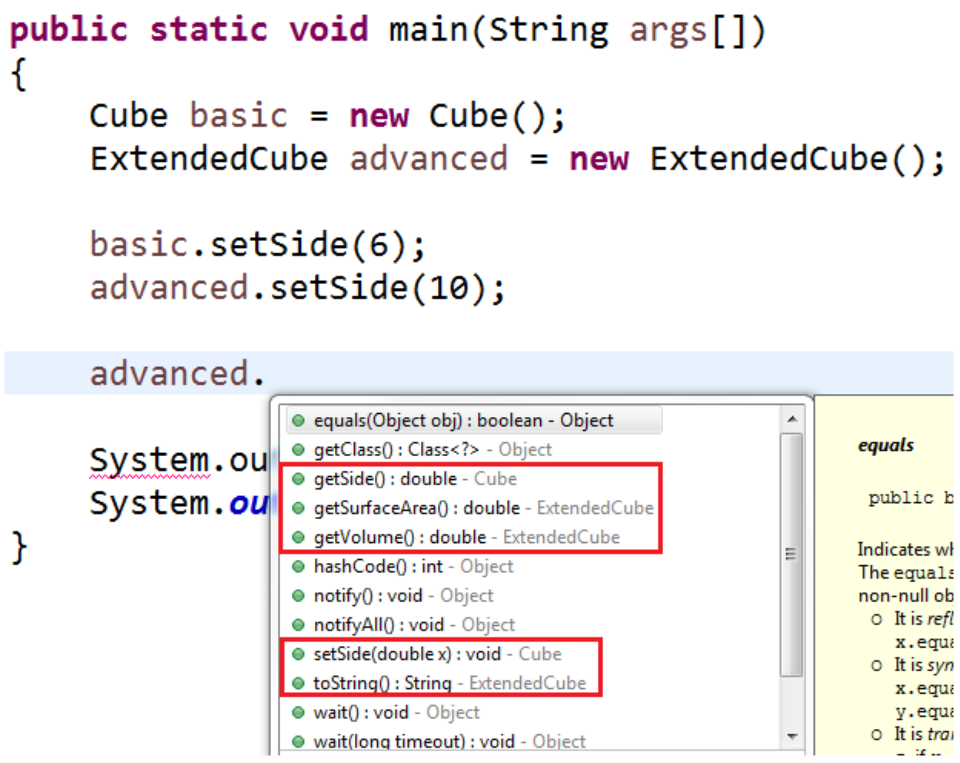


```
public static void main(String args[])
{
    Cube basic = new Cube();
    ExtendedCube advanced = new ExtendedCube();

    basic.setSide(6);
    advanced.setSide(10);

    basic.

    System
    System
}
```

equals(Object obj) : boolean - Object
getClass() : Class<?> - Object
getSide() : double - Cube
hashCode() : int - Object
notify() : void - Object
notifyAll() : void - Object
setSide(double x) : void - Cube
toString() : String - Cube
wait() : void - Object
wait(long timeout) : void - Object
wait(long timeout, int nanos) : void - Object
Press 'Ctrl+Space' to show Template Proposals
equals
public boolea
Indicates whether s
The equals metho
non-null object ref
It is reflexive: f
x.equals(x)
It is symmetric:
x.equals(y)
y.equals(x)
It is transitive:
Pres

(forgot toString)

Derived Class



```
public static void main(String args[])
{
    Cube basic = new Cube();
    ExtendedCube advanced = new ExtendedCube();

    basic.setSide(6);
    advanced.setSide(10);

    advanced.

    System.ou
    System.ou
}
```

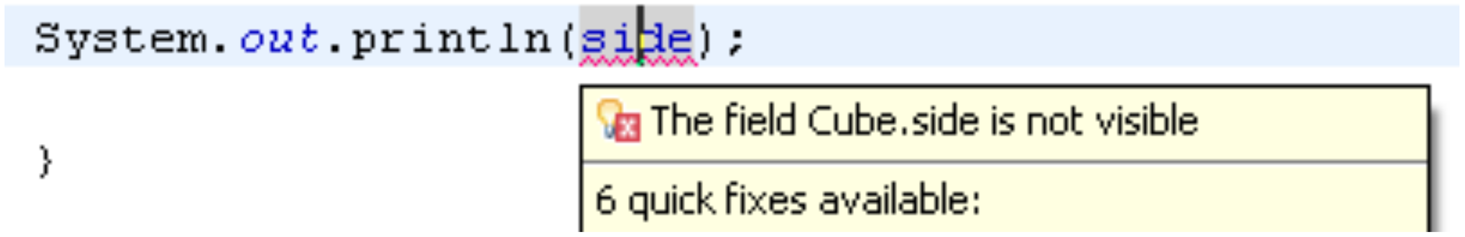
equals(Object obj) : boolean - Object
getClass() : Class<?> - Object
getSide() : double - Cube
getSurfaceArea() : double - ExtendedCube
getVolume() : double - ExtendedCube
hashCode() : int - Object
notify() : void - Object
notifyAll() : void - Object
setSide(double x) : void - Cube
toString() : String - ExtendedCube
wait() : void - Object
wait(long timeout) : void - Object
equals
public b
Indicates wh
The equals
non-null ob
It is refl
x.equa
It is sym
x.equa
y.equa
It is tra

- The IDE will also keep track of the class member's visibility
- many IDE's already help you identify which members you can reference DEPENDING on where you are physically writing code

- but you need to BIG picture in mind as well

In Cube/ExtendedCube, change data member “side” to public. What does an IDE window show now? (What does the public data member look like?)

Eclipse and automated member access



```
System.out.println(side);

}
```

The field Cube.side is not visible

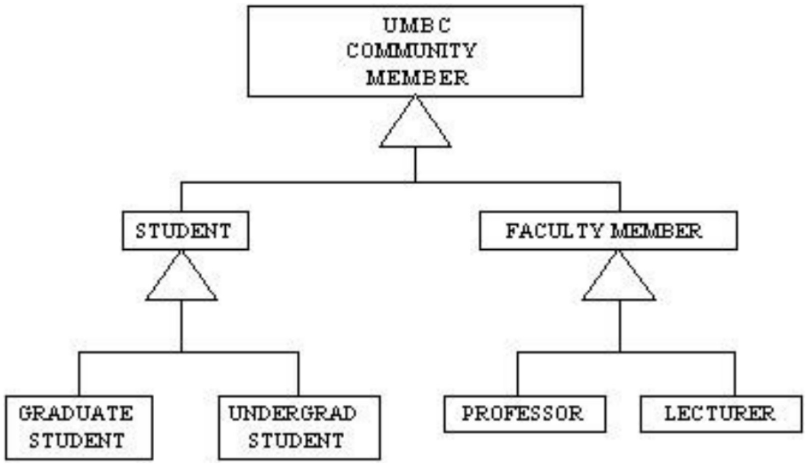
6 quick fixes available:

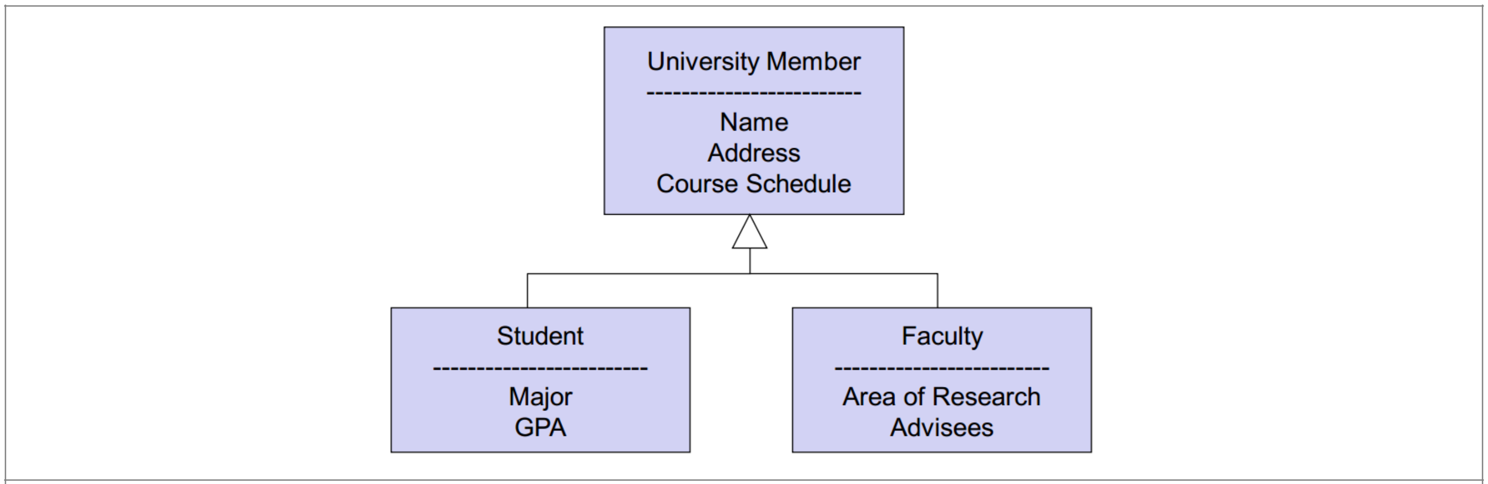
"side" is a private variable that does not have direct access and needs a getter/setter to access

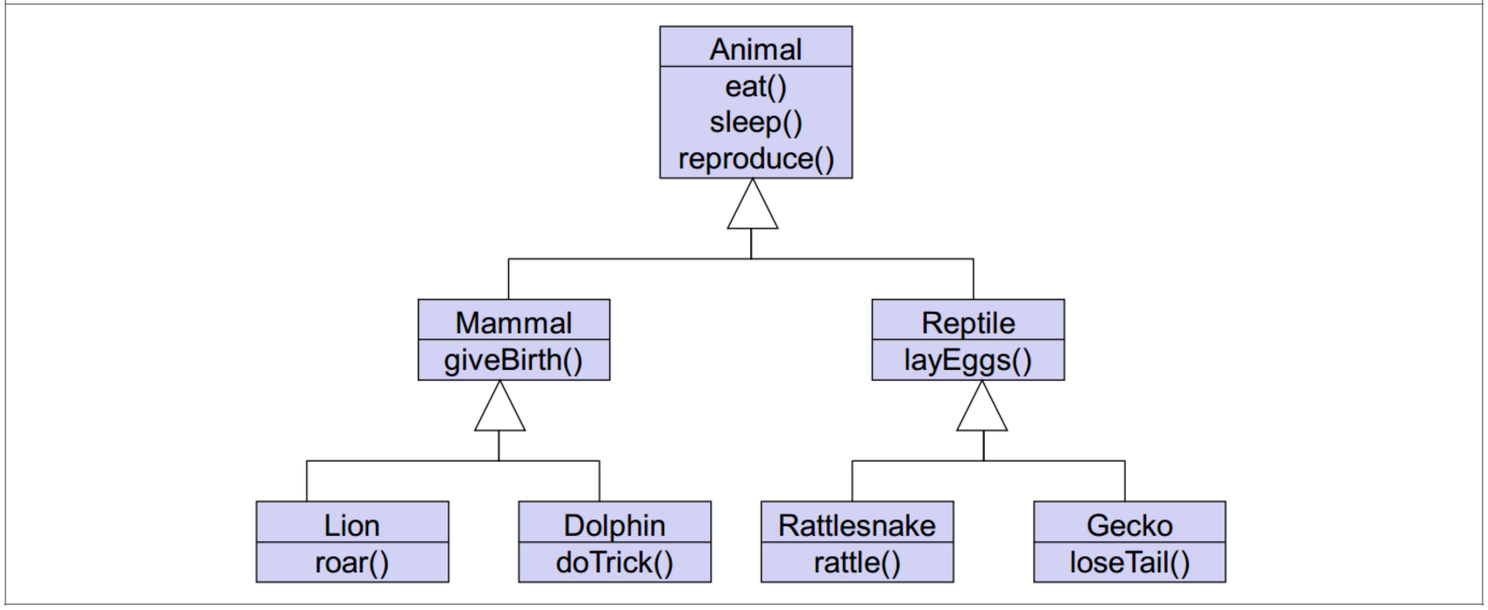
Inheritance and a game plan

- you have to be able to design and overall game plan
- start with a base object, then break into smaller more defined objects
    - the deeper (or more derived) the LESS code and material it should have

Example Inheritance Applications



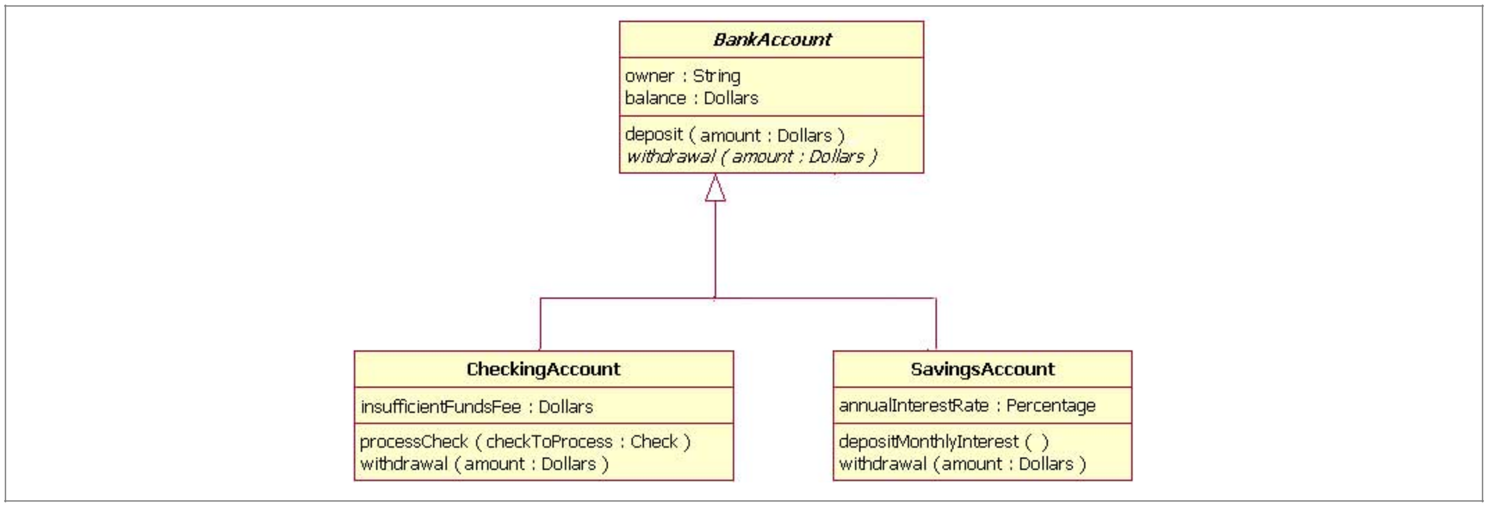




Inheritance and UML (game plan)

- So much better
- Can translate to any language

UML of Inheritance in Action


BankAccount
owner : String
balance : Dollars
deposit (amount : Dollars)
withdrawal (amount : Dollars)
CheckingAccount
insufficientFundsFee : Dollars
processCheck (checkToProcess : Check)
withdrawal (amount : Dollars)
SavingsAccount
annualInterestRate : Percentage
depositMonthlyInterest ( )
withdrawal (amount : Dollars)

The Final keyword

- the final keyword prevents an overriding **of methods and member variables, classes, etc…**
- **not a bad idea to have the base class as final**

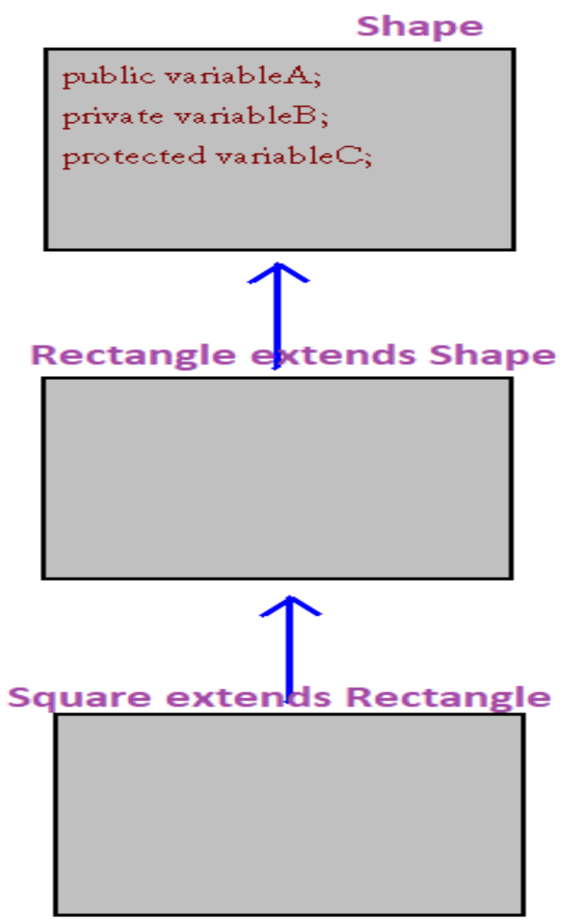
Final Example

```
public class Base
{
	private int data;

	public Base () {data = 0;}
	public Base (int n) {data = n;}

	public int getData() {return data;}
	public final String toString() { return "base's data " + data; } ← no overriding
}

public class Derived extends Base
{
	private int subData;

	public Derived (int b, int d)
	{
		super(b); // sending 'b' to super class
		subData = d;
	}

	public int getSubData() {return subData;}
	public String toString() ← Error (will not compile)
	{
		return super.toString() + "\n" +
		"subclass's data " + getSubData();
	}
}
```

Inheritance and Class Member Visibility

- public (UML +)
  - access members via the subclass class functions
- protected (UML #)
  - DIRECT access member via ONLY the subclass class functions
  - items are "shared within the family"
  - # (UML symbol)
- protected final
  - this can be the only instance of this named function/variable
- private (UML -)
  - *NOT accessible in the subclass class!*
  - UNLESS YOU USE AN ***<u>ACCESSOR METHOD!!</u>***

Class Member Visibility Example	

 Shape
public variableA;
private variableB;
protected variableC;
Rectangle extends Shape
Square extends Rectangle	

	← Can this object access: variableA? Y/N variableB? Y/N variableC? Y/N
	← Can this object access: variableA? Y/N variableB? Y/N variableC? Y/N

// Using the Cube example, in your code, try accessing Cube's side data member DIRECTLY from ExtendedCube. What does the error say??

The order of the extended classes has to be Square, then Rectangle, then Shape. Why??

BASE Object constructors and Super

- constructors work the same as before and MUST be created in the BASE class (at least)
- here are the issues
  - Base class constructor is called before Derived class constructor
  - any SUBCLASS will need to call the ***BASE CLASS's*** constructor using the "super" reserved word ***IF A BASE CLASS*** constructor is present
    - MANDITORY
- Lupoli's hints
  - start from the child's class constructor
  - but make sure the base class' constructor is satisfied

BELOW ARE LEARNING EXAMPLES ONLY!!!

(we won't really do it this way!)

Constructors and "super" in action	
Example 1 (Not OK)	

`public class Base`
`{`
`      private int data;`
 	
`      Base () {data = 0;}`
`      Base (int n) {data = n;}`
 	
`      public int getData() {return data;}`
`}`
	• base class has both a default and programmer defined constructor
	• the sub-class does not have a constructor, so by default calls the base class' default constructor
	• this set up is NOT ***OK***

```
public class Derived extends Base
{
	private int subData;

	public int getSubData() {return subData;}

}
```

```
Derived example1 = new Derived(); // will work
Derived example2 = new Base(5); // will not work, Derived MUST have a constructor with one parameter
Derived example3 = (Derived) new Base(5); // will not work, Derived MUST have a constructor with one
parameter
Derived example4 = new Derived(3,4); // will not work, Derived MUST have a constructor with one
parameter
```

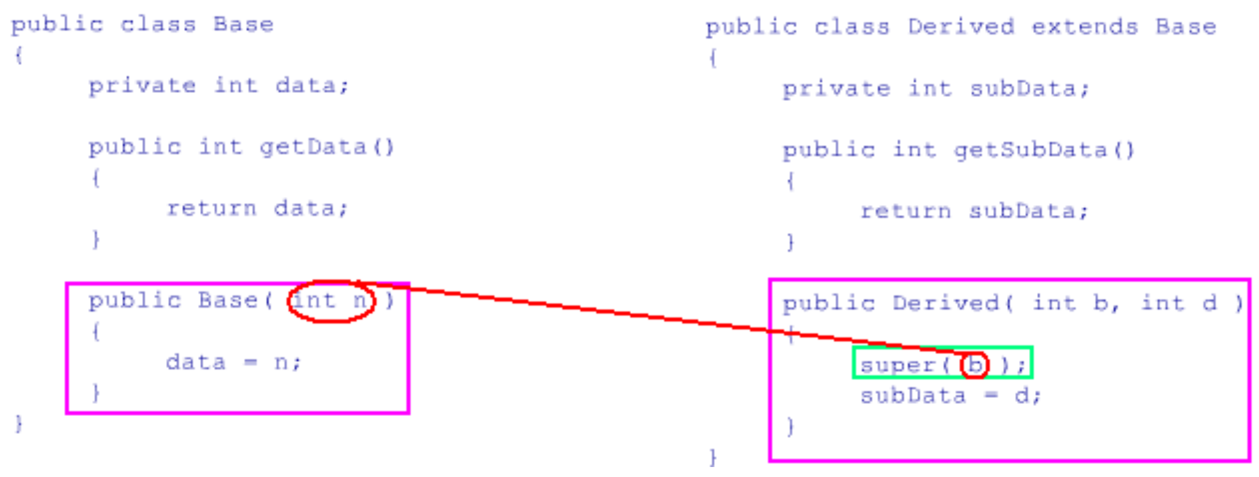
Example 2 (Not OK)

```java
public class Base
{
	private int data;

	public Base (int n) {data = n;}

	public int getData() {return data;}
}
```

```java
public class Derived extends Base
{
	private int subData;

	public int getSubData() {return subData;}

}
```

- the base class has a programmer defined constructor
- this will not compile since the base class has a defined constructor, but the subclass does not call the base class' constructor
- this setup is ***not OK***

Example 3 (OK)



```java
public class Base
{
	private int data;

	public int getData()
	{
		return data;
	}

	public Base( int n )
	{
		data = n;
	}
}
```

```java
public class Derived extends Base
{
	private int subData;

	public int getSubData()
	{
		return subData;
	}

	public Derived( int b, int d )
	{
		super( b );
		subData = d;
	}
}
```

- the base class is the same
- the subclass specifically calls the base class constructor using the "super" command
- notice the parameter in the super is the same as the number of parameters in the base class

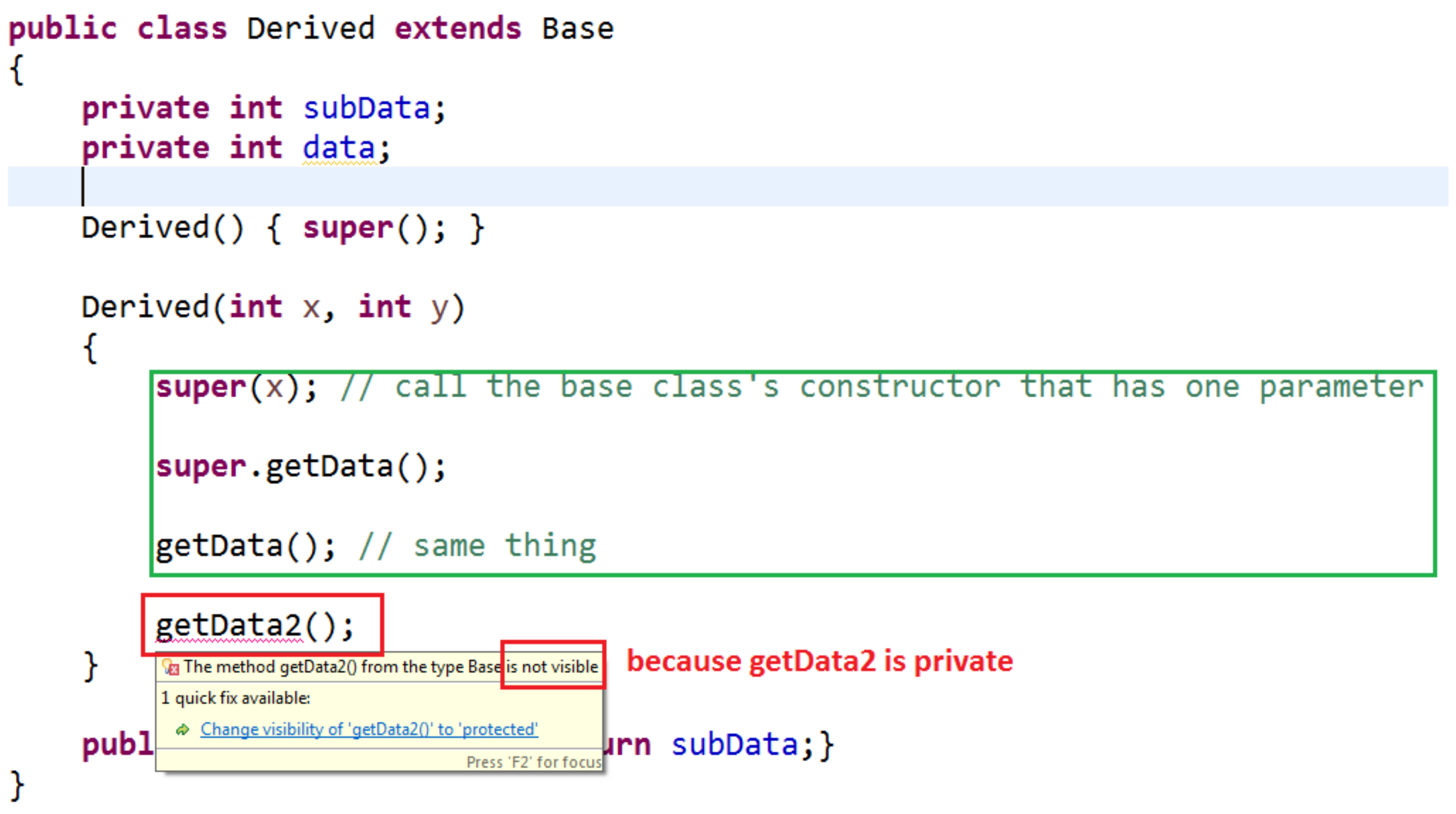
```java
Derived example1 = new Derived(); // will not work, Derived does not have a constructor with no parameters
Derived example2 = new Derived(5); // will not work, Derived does not have a constructor with one parameter
Derived example3 = new Derived(3,4); // will work
```

Super calls in a nutshell

- Only in a derived class can call “super” IF the method/member variable is public or protected!!
  - Not private!!

Super calls

```
public class Base
{
     public int data;
     protected double price = 23.23;

     Base () {data = 0;}
     Base (int n)
     {
          data = n;
     }

     public int getData() {return data;}
     private int getData2() {return data;}
}
```



```
public class Derived extends Base
{
    private int subData;
    private int data;

    Derived() { super(); }

    Derived(int x, int y)
    {
        super(x); // call the base class's constructor that has one parameter

        super.getData();

        getData(); // same thing

        getData2();
    }

    publ                                              urn subData;}
}
```

The method getData2() from the type Base is not visible

1 quick fix available:

Change visibility of 'getData2()' to 'protected'

Press 'F2' for focus

because getData2 is private

Using the Cube/Extended Cube, the overall goal is to have ExtendedCube’s constructor pass the side value. Make sure to have a “super” call in ExtendedCube’s constructor.

Game planning

- In proper planning using inheritance start with an application that could use inheritance to stop redundancy of code
- Start drawing out the various objects of the application first and list their respective members

The various Vehicles						
Plane	**Truck**	**Submarine**	**Motorcycle**	**Spaceship**	**Car**	**Train**
wingSpan engineType numberOfSeats speed numberOfPilots numberOfPassengers fuelType autopilot wheels	Gas tank Seats Radio Gears Tire wheel Doors Weight License Engine (V6) Price Electric A/C Fluid Key Mirror	Flag (operating country) Manufacturer Year of construction Type (military, civilian,...) Operating range Sonar Range Maximum speed Maximum bottom time Min operating crew Max crew capacity Power source Armament	Speedometer Breaks Gas Tank Handlebar VIN # Clutch Wheels Radio Engine	doors brakes boosters lazers lights steering VIN wigns engine seats windows saftey_eq pilot cargo electronics comm_eq	Weight Wheels Fuel Seats Electronics License Plate VIN # Horn Windows Doors Lights A/C Fluids Gears Steering Cargo	TicketCost Cargo Dinning Beds Tracks Wheels Doors Carpet Charis Crew TypeTrain PowerSource Whistle Light Breaks

- Group related members into a base class (in theory)
  - This will take some adjustments and "compromises"
  - The member name may need to be more detailed

The various Vehicles (after combining like members)

Plane	Truck	Submarine	Motorcycle	Spaceship	Car	Train
wingSpan Type numberOfPilots fuelType autopilot	Gas tank Radio Gears Tire wheel Price Electric A/C Fluid Key Mirror	Flag (operating country) Type (military, civilian,...) Operating range Sonar Range Maximum bottom time Min operating crew Max crew capacity Power source Armament	Brakes Gas Tank Clutch Radio	brakes boosters lazers lights steering saftey_eq pilot cargo hyperdrive electronics comm_eq	Fuel Electronics Horn Lights A/C Fluids Gears Steering Cargo	TicketCost Cargo Dinning Beds Tracks Carpet Charis Crew TypeTrain Whistle Light Breaks

The new BASE class

Vehicle Class

```
// overall vehicle details
float weight;
int yearMade; // would love to use short, but would have to cast
String make;
String model;
String color;
String ID;
int wheels;

// people related
int occupants;

// compartment related
int bathrooms;
int seats;
int doors;
int windows;

// engine related
String powerSource;
String engineType;
int horsePower;
int engineCount;
```

- Determine accessibility
  - Base class
    - Data members in the BASE should be **protected** at least
      - Validation, but shared among it's derived classes
    - functions should be **protected** at least

The new BASE class
Vehicle Class

```
// overall vehicle details
- float weight;
- int yearMade; // would love to use short, but would have to
cast
- String make;
- String model;
- String color;
- String ID;
- int wheels;

// people related
- int occupants;

// compartment related
- int bathrooms;
- int seats;
- int doors;
- int windows;

// engine related
- String powerSource;
- String engineType;
- int horsePower;
- int engineCount;
```

- Derived class
  - Data members should be private
  - functions should be public
  - So the main can access

The various Vehicles (after combining like members)						
Plane	**Truck**	**Submarine**	**Motorcycle**	**Spaceship**	**Car**	**Train**
− wingSpan − Type − numberOfPilots − fuelType − autopilot	− Gas tank − Radio − Gears − Tire wheel − Price − Electric − A/C − Fluid − Key − Mirror	− Flag (operating country) − Type (military, civilian,...) − Operating range − Sonar Range − Maximum bottom time − Min operating crew − Max crew capacity − Power source − Armament	− Brakes − Gas Tank − Clutch − Radio	− brakes − boosters − lazers − lights − steering − saftey_eq − pilot − cargo − hyperdrive − electronics − comm_eq	− Fuel − Electronics − Horn − Lights − A/C − Fluids − Gears − Steering − Cargo	− TicketCost − Cargo − Dinning − Beds − Tracks − Carpet − Charis − Crew − TypeTrain − Whistle − Light − Breaks

- Code the BASE class
    - Create the data members (private)
    - Use all IDE features to build the rest of the
        - Mutators/accessors (protected) and toString
        - Constructors (Fully Programmer defined)

Code the Base Class first

```
public class Vehicle
{
        // overall vehicle details
        private float weight;
        private int yearMade; // would love to use short, but would have to cast
        private String make;
        private String model;
        private String color;
        private String ID;
        private int wheels;

        // people related
        private int occupants;

        // compartment related
        private int bathrooms;
        private int seats;
        private int doors;
        private int windows;

        // engine related
        private String powerSource;
        private String engineType;
        private int horsePower;
        private int engineCount;

        // velocity related
        private int maxSpeed;

        // constructor
        public Vehicle(float weight, int yearMade, String make, String model, String color, String iD, int wheels,
int i,
                        int j, int k, int l, int m, String powerSource, String engineType, int horsePower,
                        int engineCount, int n) {
                this.weight = weight;
                this.yearMade = yearMade;
                this.make = make;
                this.model = model;
                this.color = color;
                ID = iD;
                this.wheels = wheels;
                this.occupants = i;
                this.bathrooms = j;
                this.seats = k;
                this.doors = l;
                this.windows = m;
                this.powerSource = powerSource;
                this.engineType = engineType;
                this.horsePower = horsePower;
                this.engineCount = engineCount;
                this.maxSpeed = n;
        }

        // getters and setter (Mutators and Accessors)
        protected float getWeight() { return weight; }
        protected void setWeight(float weight) { this.weight = weight; }
        protected int getYearMade() { return yearMade; }
        protected void setYearMade(int yearMade) { this.yearMade = yearMade; }
        protected String getMake() { return make; }
        protected void setMake(String make) { this.make = make; }
        protected String getModel() { return model; }
        protected void setModel(String model) { this.model = model; }
        protected String getColor() { return color; }
        protected void setColor(String color) { this.color = color; }
        protected String getID() { return ID; }
        protected void setID(String iD) { ID = iD; }
        protected int getWheels() { return wheels; }
        protected void setWheels(int wheels) { this.wheels = wheels; }
```

- Code the Derived class
  - Create the data members
  - Use all IDE features to build the rest of the
    - Mutators/accessors (public)
  - In the Derived class, code the Extend to it’s BASE class
    - **This updates the IDE to let it know your derived class has inherited features**
    - Create constructor
      - Should have super(….) in it now to fill the base class members
    - Create toString

Creating the Derived class

Before Extends Vehicle

```
public class Plane {

        int wingSpan = 0;
        int numberOfPilots = 0;
        String fuelType;
        boolean autoPilot;

        public int getWingSpan() { return wingSpan; }
        public int getNumberOfPilots() { return numberOfPilots; }
        public String getFuelType() { return fuelType; }
        public boolean isAutoPilot() { return autoPilot; }
        public void setWingSpan(int wingSpan) { this.wingSpan = wingSpan; }
        public void setNumberOfPilots(int numberOfPilots) { this.numberOfPilots = numberOfPilots; }
        public void setFuelType(String fuelType) {this.fuelType = fuelType; }
        public void setAutoPilot(boolean autoPilot) { this.autoPilot = autoPilot;}

}
```

After Extends Vehicle

```java
public class Plane extends Vehicle{

	int wingSpan = 0;
	int numberOfPilots = 0;
	String fuelType;
	boolean autoPilot;


	public Plane(float weight, int yearMade, String make, String model,
			String color, String iD, int wheels, int i, int j, int k, int l,
			int m, String powerSource, String engineType, int horsePower,
			int engineCount, int n, int wingSpan, int numberOfPilots,
			String fuelType, boolean autoPilot) {
		super(weight, yearMade, make, model, color, iD, wheels, i, j, k, l, m,
				powerSource, engineType, horsePower, engineCount, n);
		this.wingSpan = wingSpan;
		this.numberOfPilots = numberOfPilots;
		this.fuelType = fuelType;
		this.autoPilot = autoPilot;
	}

	public int getWingSpan() { return wingSpan; }
	public int getNumberOfPilots() { return numberOfPilots; }
	public String getFuelType() { return fuelType; }
	public boolean isAutoPilot() { return autoPilot; }
	public void setWingSpan(int wingSpan) { this.wingSpan = wingSpan; }
	public void setNumberOfPilots(int numberOfPilots) { this.numberOfPilots = numberOfPilots; }
	public void setFuelType(String fuelType) {this.fuelType = fuelType; }
	public void setAutoPilot(boolean autoPilot) { this.autoPilot = autoPilot;}

	@Override
	public String toString() {
		return "Plane [wingSpan=" + wingSpan + ", numberOfPilots="
				+ numberOfPilots + ", fuelType=" + fuelType + ", autoPilot="
				+ autoPilot + ", getWeight()=" + getWeight()
				+ ", getYearMade()=" + getYearMade() + ", getMake()="
				+ getMake() + ", getModel()=" + getModel() + ", getColor()="
				+ getColor() + ", getID()=" + getID() + ", getWheels()="
				+ getWheels() + ", getOccupants()=" + getOccupants()
				+ ", getBathrooms()=" + getBathrooms() + ", getSeats()="
				+ getSeats() + ", getDoors()=" + getDoors() + ", getWindows()="
				+ getWindows() + ", getPowerSource()=" + getPowerSource()
				+ ", getEngineType()=" + getEngineType() + ", getHorsePower()="
				+ getHorsePower() + ", getEngineCount()=" + getEngineCount()
				+ ", getMaxSpeed()=" + getMaxSpeed() + ", toString()="
				+ super.toString() + ", getClass()=" + getClass()
				+ ", hashCode()=" + hashCode() + "]";
	}

}
```

- Create the Driver Class
  - Create one instance to test

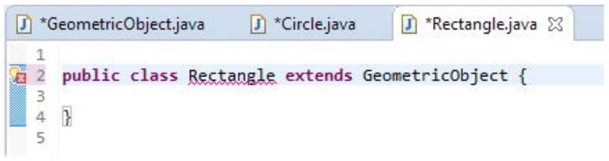
Create an instance in the Driver

```
public class Driver {
      public static void main(String[] args) {

            Plane SouthWest = new Plane(10000, 2014, "Boeing", "787", "Orange", "SW
1976",
                        5, 150, 3, 150, 7, 55, "Jet Fuel (JP8)", "Pratt and
Whitney", 2500,
                        4, 400, 80, 2, "Jet Fuel (JP8)", false);

            System.out.println(SouthWest);
      }
}
```

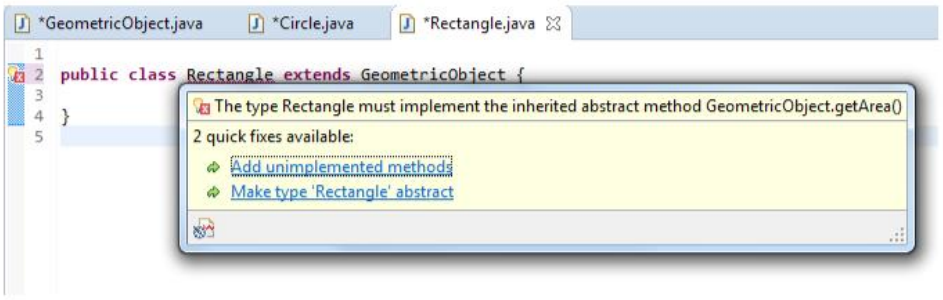
Building subclass functions quickly

- thanks Nick Levandoski UMBC training Su'14
- after the creation of your base class, any functions that also need to be addressed in subclass can be automatically generated
  - again, the base class must be completed first

Lazy Eclipse Users (sub class generation)



```
public class Rectangle extends GeometricObject {

}
```



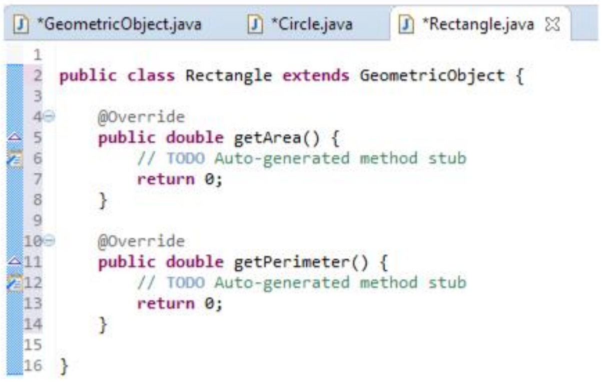
```
public class Rectangle extends GeometricObject {
}
```

The type Rectangle must implement the inherited abstract method GeometricObject.getArea()

2 quick fixes available:

Add unimplemented methods

Make type 'Rectangle' abstract



```
public class Rectangle extends GeometricObject {

    @Override
    public double getArea() {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public double getPerimeter() {
        // TODO Auto-generated method stub
        return 0;
    }

}
```

Overriding (not overloading, but darn close)

- same as overloading, but between related objects (base – subclass)
  - in C++ this would be done using the “virtual” keyword in the base class
- subclass has a method with the same **name and number and (possibly) type** of parameters as a method in a super class
  - notice, same number/type of parameters (not like overloading)
- use must also worry about member method/member visibility
  - a private in a base class cannot be accessed by a subclass

Overriding Example

```
public class Base
{
	private int data;

	public Base () {data = 0;}
	public Base (int n) {data = n;}

	public int getData() {return data;}
	public String toString() { return "base's data " + data; }
}

public class Derived extends Base ← Subclass inherited the toString() method
{
	private int subData;

	public Derived (int b, int d)
	{
		super(b); // sending 'b' to super class
		subData = d;
	}

	public int getSubData() {return subData;}
	public String toString()← Subclass overrides the inherited method
	{
		return super.toString() + "\n" + ← invokes the inherited the toString() method
		"subclass's data " + getSubData();
	}
}
```

```
Derived example3 = new Derived(3,4); // will work
System.out.println(example3);
```

```
base's data 3
subclass's data 4
```

- notice two display methods
- also notice subclass' calls base class' (super) display
- *A TOSTRING method is much better than a display method (just used this example)

Rules for overriding an Inherited Methods

- the overriding method's header (signature) must be exactly the same as the header of the inherited method
- the visibility of the overriding method cannot be more restrictive than the visibility of the inherited method
- constructors ***cannot*** be overridden
- a function with static ***cannot*** be overridden

Overriding vs. Overloading

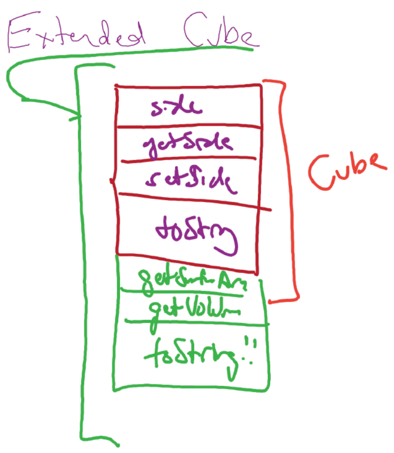
- Overriding
    - Same function name or "signature" in 2 or more inherited (related) objects
- Overloading
    - Same function name or "signature" in 1 object **(except parameters)**

Developing Inheritance Structures Guide

- Look for Objects that have common attributes and behaviors
- Factor common items into a single class
- Determine which subclasses need specialized features
- Examine groups of subclasses that may exhibit common behaviors
- Apply the IS-A test

// Develop a structure for a Contestant, then it's child classes Bowler and Golfer

Solutions

ExtendedCube overview
 Extended Cube side getSide setSide toString Cube getSurfArea getVolume toString!!

Sources

UMBC CMSC 202 Notes

http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/