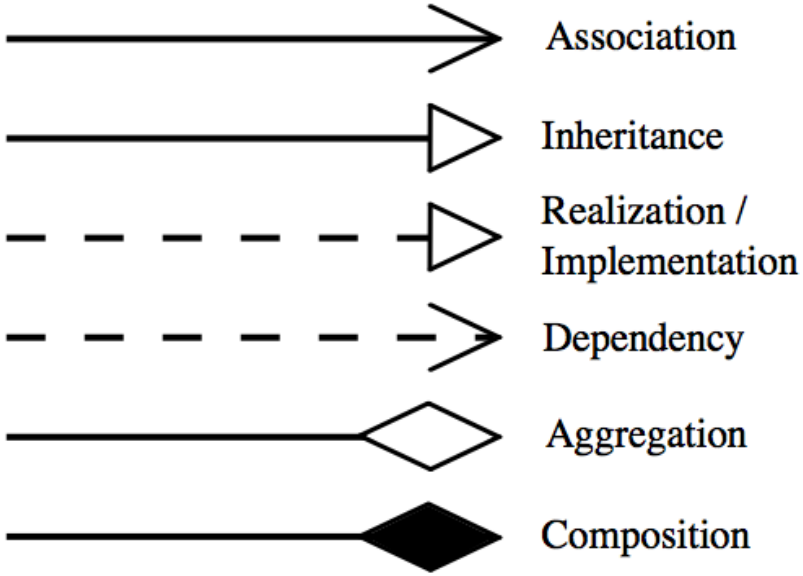
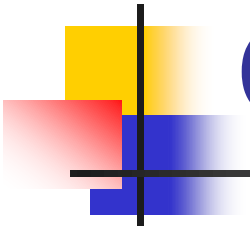
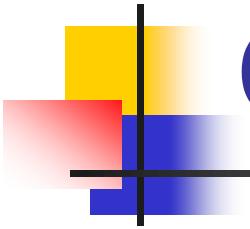
04. OO Relationships





OO Relationships

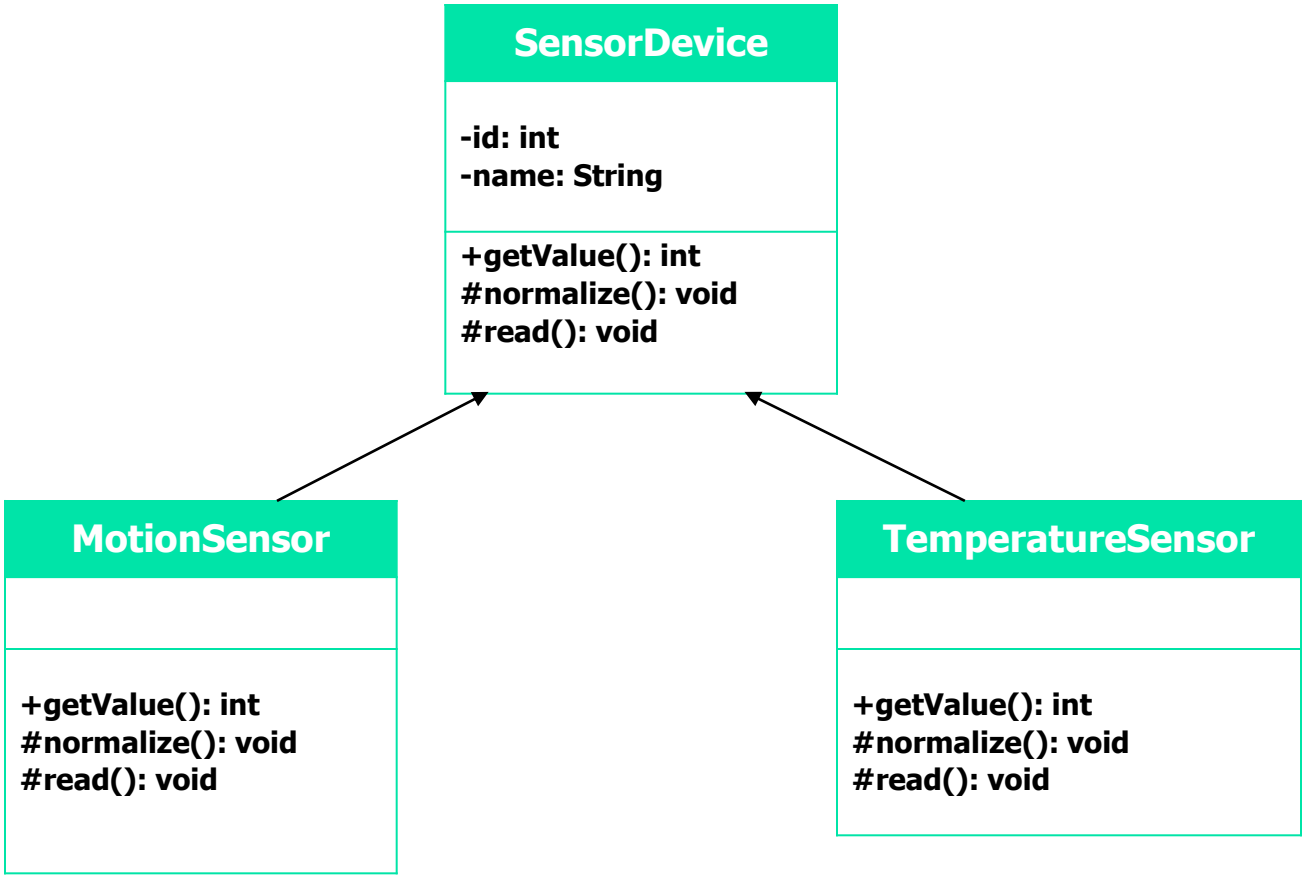
- Associations
  - Indicate that instances of one model element are connected to instances of another model element
- Generalizations
  - Indicate that one model element is a specialization of another model element

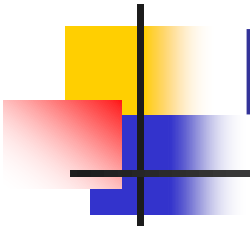


OO Relationships

- Realizations
  - Indicate that one model element provides a specification that another model element implements
- Dependencies
  - Indicate that a change to one model element can affect another model element

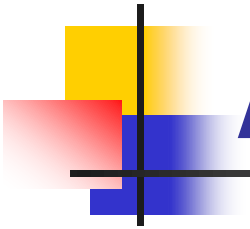
Inheritance





Dynamic Method Dispatch

- Dynamic method dispatch is the mechanism by which a call to an overridden function is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how java implements run-time polymorphism.



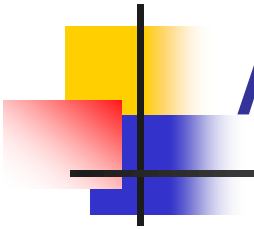
protected access modifier

- The *protected* access modifier is used for fields or methods and cannot be used for classes and Interfaces.
- It also cannot be used for fields and methods within an interface.
- Fields, methods and constructors declared protected in a superclass can be accessed only by its subclasses.
- Classes in the same package can also access protected fields, methods and constructors as well, even if they are not a subclass of the protected member's class.

abstract modifier

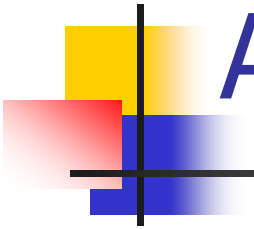
- We declare a class *abstract* when we want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- That is, when a superclass is unable to create a meaningful implementation for a method.
- The *abstract* modifier can be applied to classes and methods.

Abstract Class



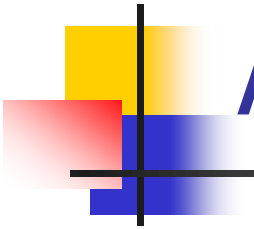
- An abstract class cannot be instantiated.
- Abstract classes provide a way to defer implementation to subclasses.
- Declaration:

```
abstract class MyClass  {
        . . .
}
```



Abstract Method

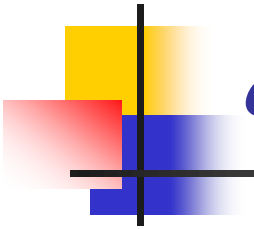
- No implementation for a method. Only the signature of the method is declared.
- Used to put some kind of compulsion on the person who inherits from this class. i.e., the person who inherits **MUST** provide the implementation of the method to create an object.
- A method can be made abstract to defer the implementation. i.e., when you design the class, you know that there should be a method, but you don't know the algorithm of that method.



Abstract Method

- Declaration:

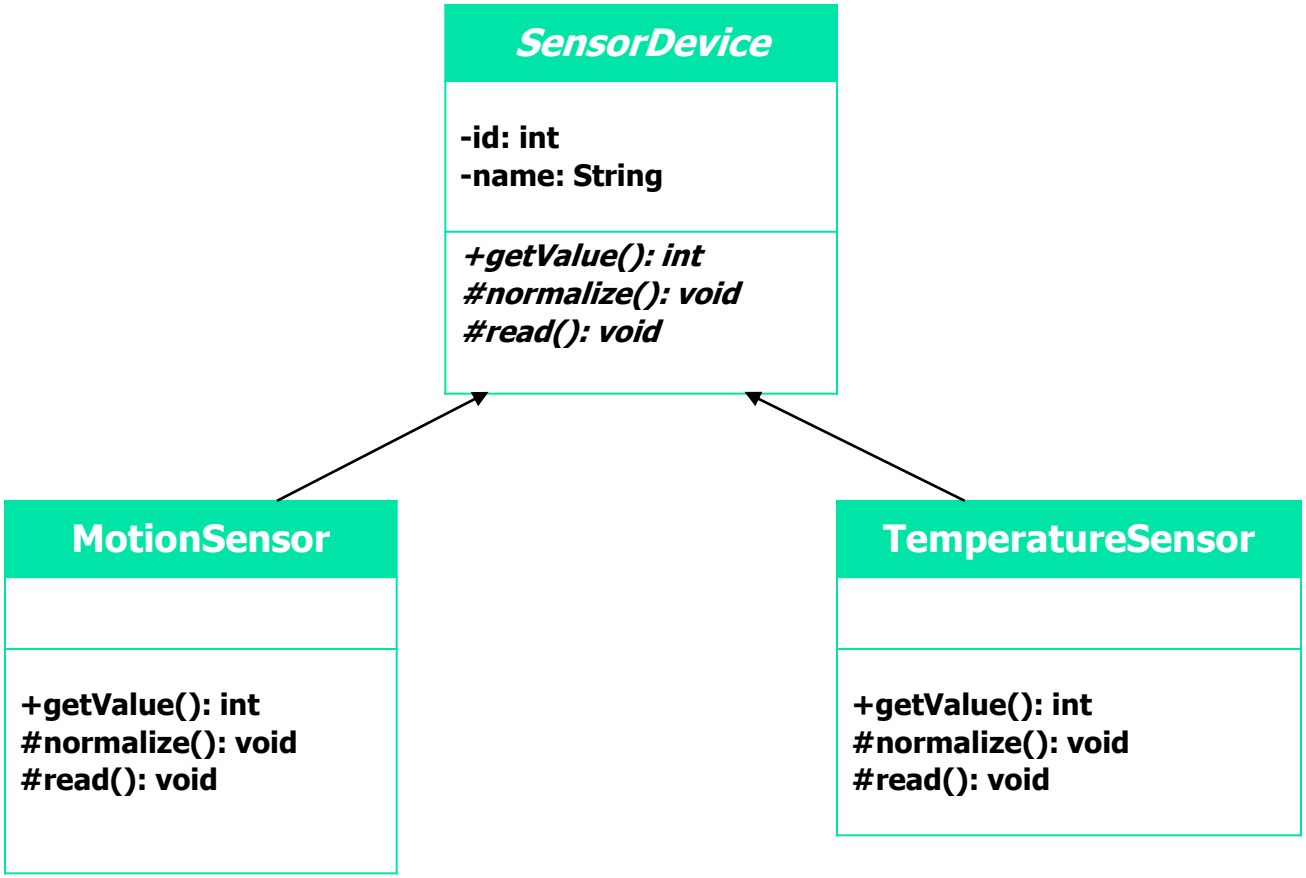
  abstract void myMethod();



abstract modifier

- A class **must** be declared *abstract* if any of the following conditions is true:
  - The class has one or more abstract methods.
  - The class inherits one or more abstract methods (from an abstract parent) for which it does not provide implementations.

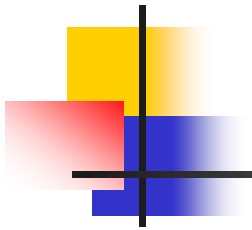
Inheritance



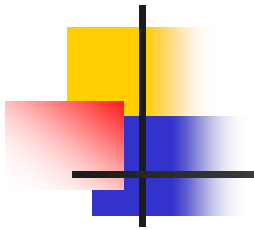
Example – Abstract class

```
Abstract class Shape  {
    double dim1;
    double dim2;

    Shape(double a, double b)  {
            dim1 = a;
            dim2 = b;
    }

    abstract double area();
}
```

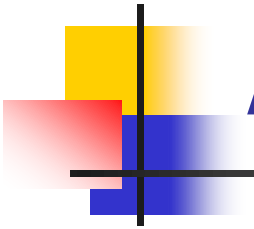
```
class Rectangle extends Shape  {

    Rectangle(double a, double b)  {
            super(a, b);
    }

    double area()  {
            System.out.println("Inside
                                rectangle");
            return dim1*dim2;
    }
}
```



```
class Triangle extends Shape  {

    Triangle(double a, double b)  {
            super(a, b);
    }

    double area()  {
            System.out.println("Inside triangle");
            return dim1*dim2/2;
    }
}
```

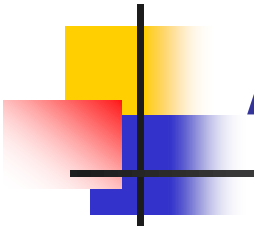


```
class AreaFinder  {

    public static void main(String args[])  {

        // Shape f = new Shape(10, 10);  // illegal now.
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);

        Shape ref;

        ref = r;
        System.out.println("Area is " + ref.area());

        ref = t;
        System.out.println("Area is " + ref.area());
    }
}
```



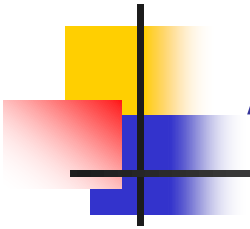
final modifier

- The *final* modifier can be applied to variables, methods, and classes.



final variables

- A variable can be declared as *final*.
- Doing so prevents its contents from being modified.
- We must initialize a *final* variable when it is declared. (***final*** ≈ ***const*** in C / C++ / C#)

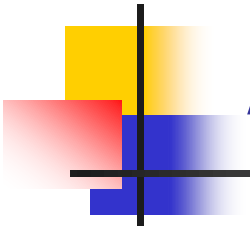


final variables

Example:

```
final int FILE_NEW = 1;
final double PI = 3.142857;
```

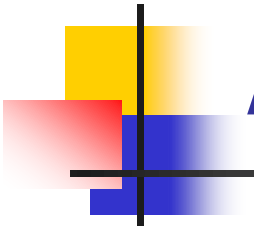
- It is common coding convention to use all uppercase letters for final variables.
- Variables declared as final do not occupy memory on a per-instance basis.



final methods

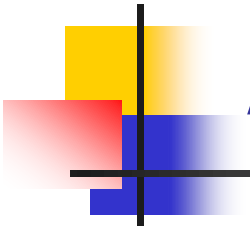
- Methods declared as final cannot be overridden.

```
class A  {
    final void myMethod() {
            System.out.println("This is a final method");
    }
}

class B extends A  {
    void myMethod()  {  // ERROR! Cannot Override.
            System.out.println("Illegal");
    }
}
```



final classes

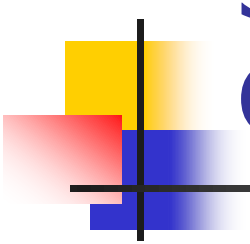
- Used to prevent a class from being inherited.
- Declaring a class final implicitly declares all of its methods as final too.
- It is illegal to declare a class as both abstract and final.



final classes

- Example:

```
final class A  {
      . . .
}

class B extends A  {  // ERROR! Can't subclass A.
      . . .
}
```



Summary: **Classes and Abstract Classes**

- Classes
  - All declared methods must be defined.
  - No restriction on member variables.
- Abstract Classes
  - Some methods may be defined.
  - No restriction on member variables.