Object Oriented Thoughts

A paradigm shift from procedural to object-based programming

Md Saeed Siddik

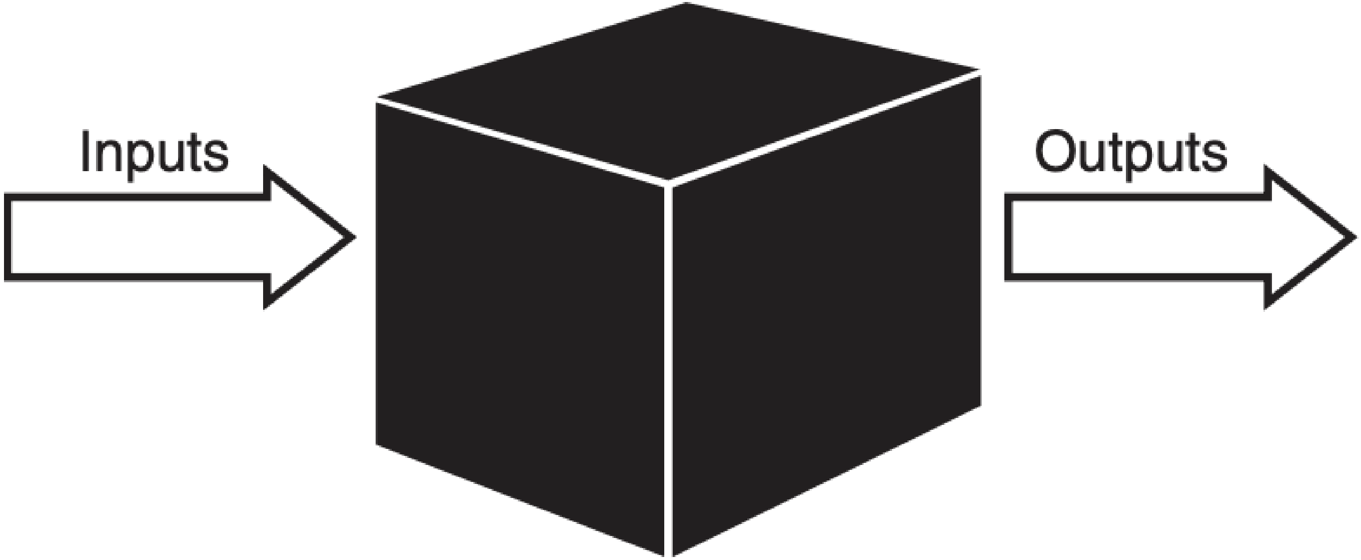
Assistant Professor,

IIT University of Dhaka

Why do we need the object-based thinking?

- For modeling real-world problems
- For better code organization and reuse
- For easier maintenance and scalability

Procedural Versus OO Programming



Object forms with attributes and behaviors

- An object is an entity that contains both data and behavior.
- Behaviors are contained in methods
- Attributes are contained in variables

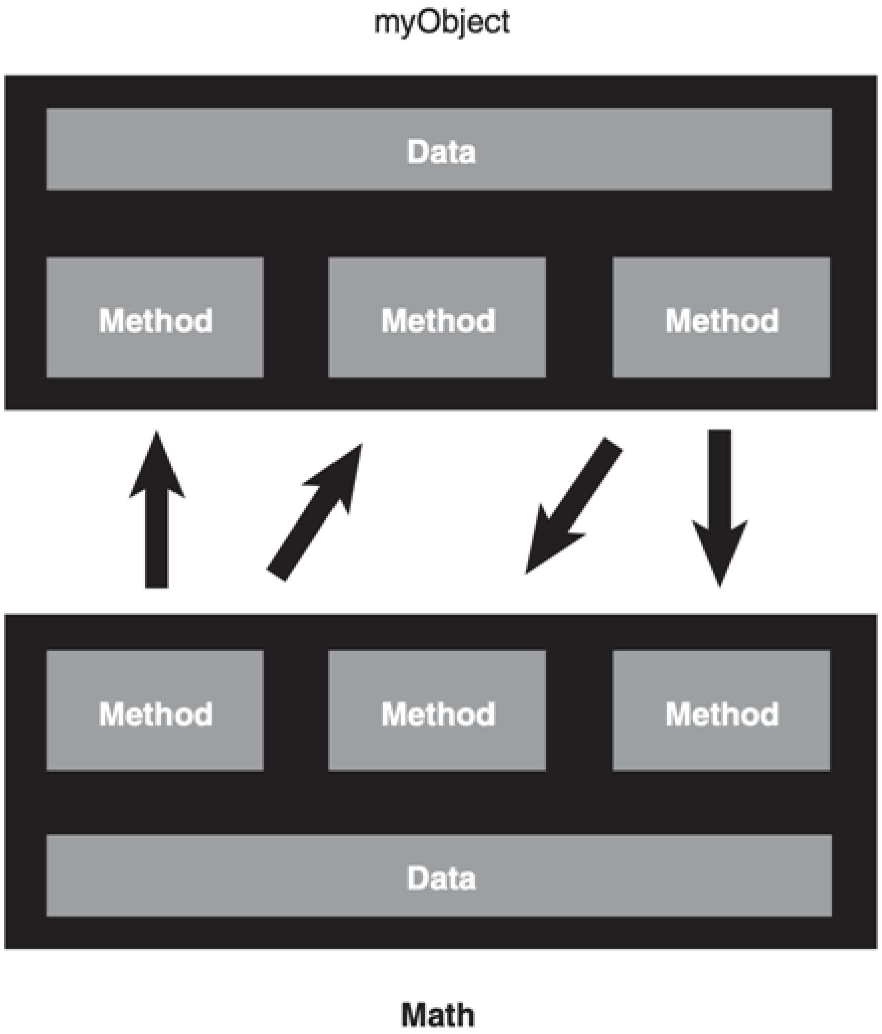
- A person has attributes, such as eye color, age, height, and so on.
- A person also has behaviors, such as walking, talking, breathing, and so on.

- In **OO design,** the attributes and behaviors are contained within a **single object,** whereas in procedural, or structured design, the attributes and behaviors are normally **separated**.

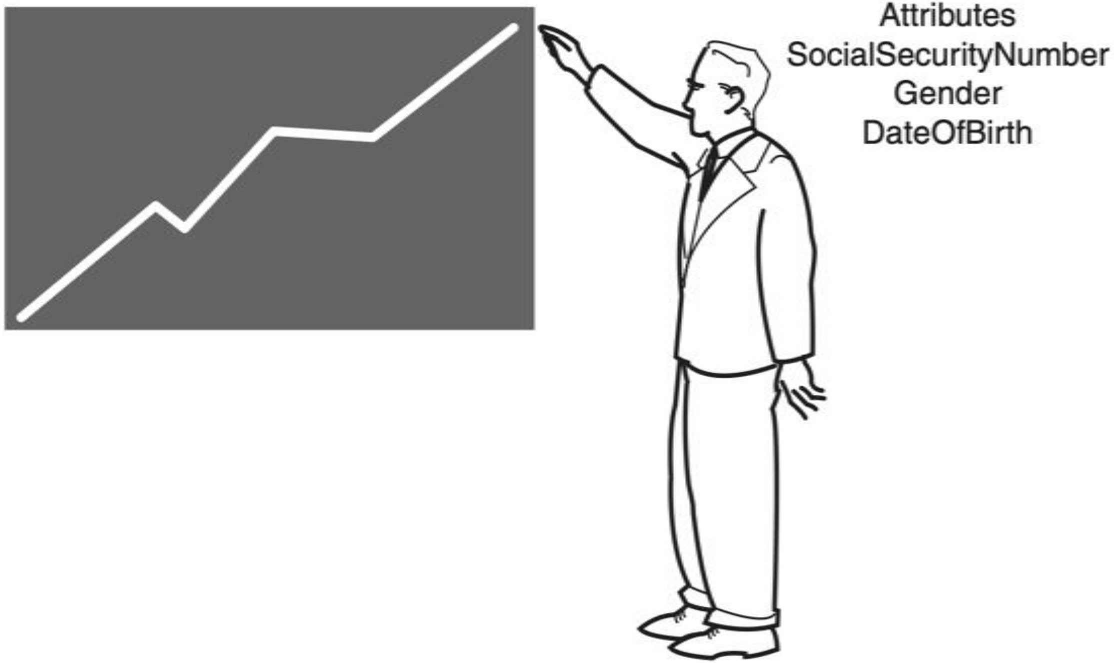
Let’s discuss some examples of Objects

Class Design Guidelines

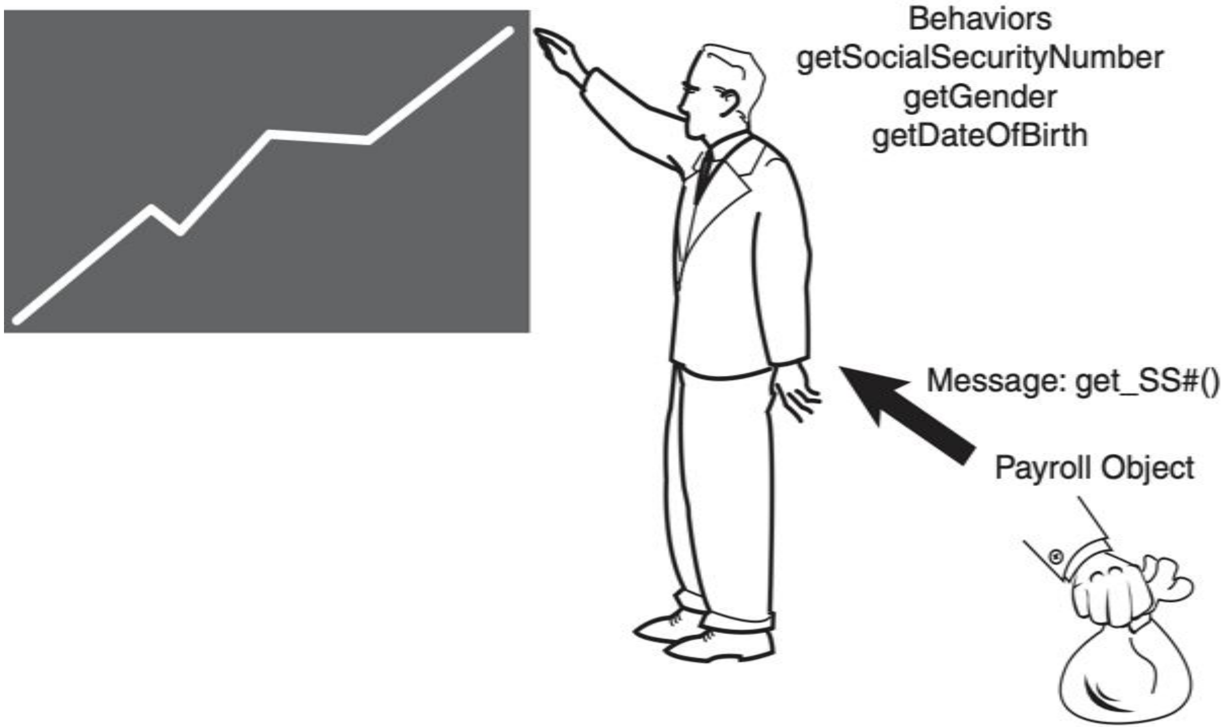
- Data Hiding
- Communication



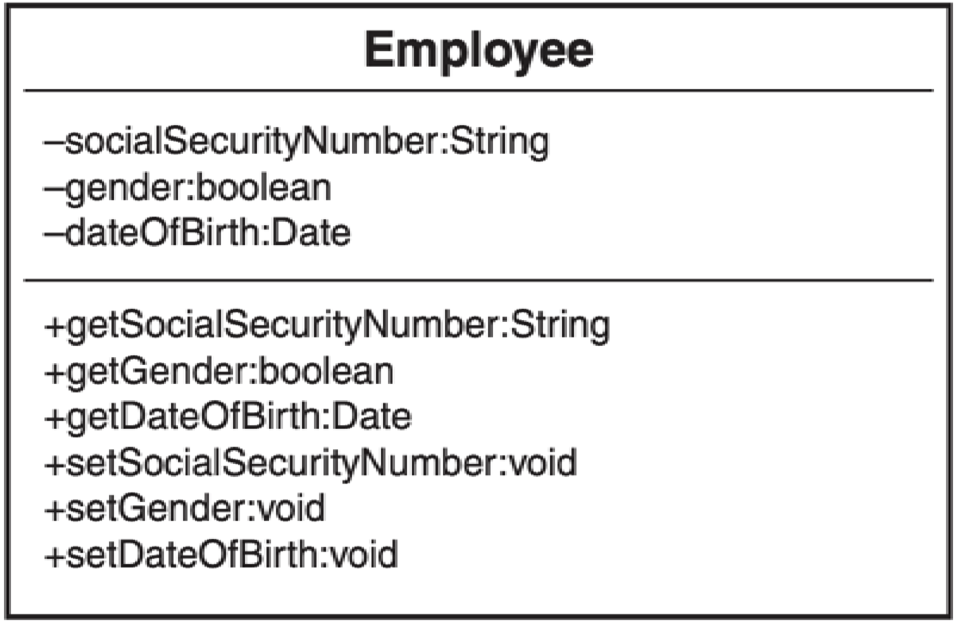
How to communicate with Objects?



How to communicate with Objects?



UML Class Diagrams

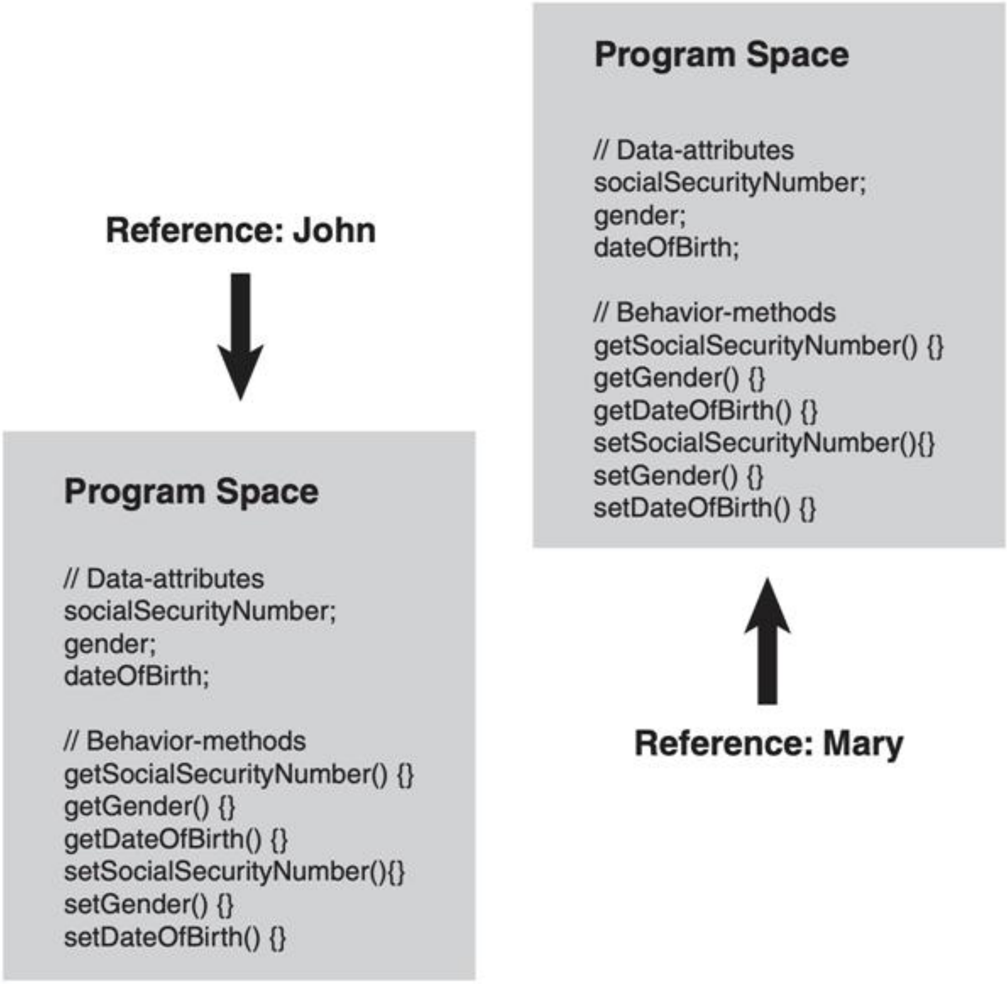


Employee

–socialSecurityNumber:String
–gender:boolean
–dateOfBirth:Date

+getSocialSecurityNumber:String
+getGender:boolean
+getDateOfBirth:Date
+setSocialSecurityNumber:void
+setGender:void
+setDateOfBirth:void

Program Space in OOP



Reference: John

Program Space

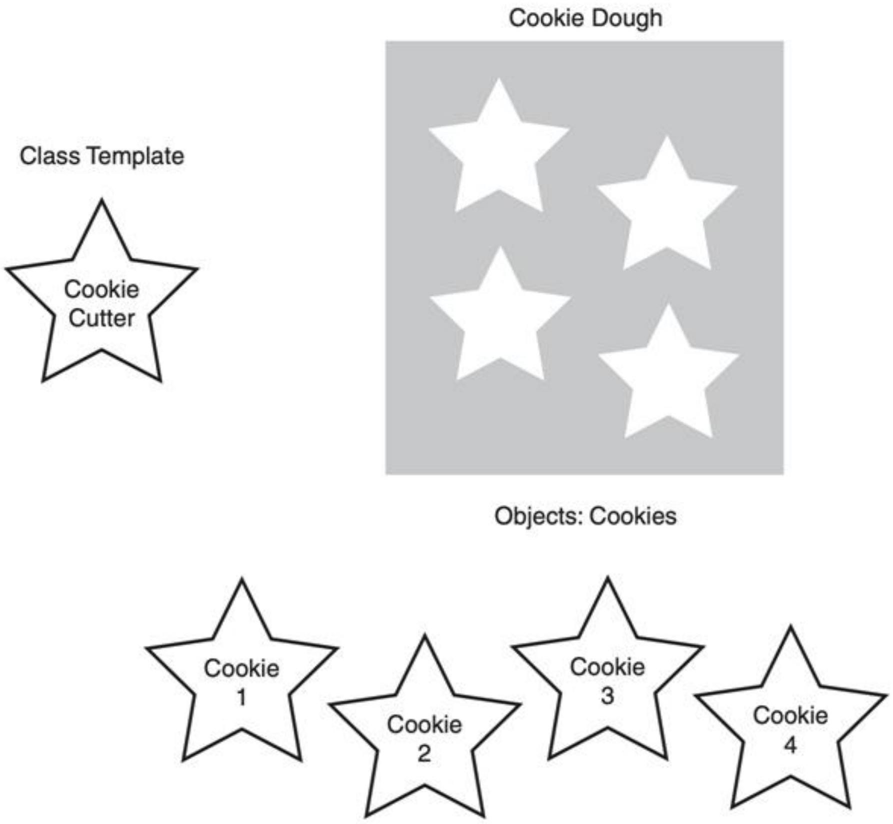
```
// Data-attributes
socialSecurityNumber;
gender;
dateOfBirth;

// Behavior-methods
getSocialSecurityNumber() {}
getGender() {}
getDateOfBirth() {}
setSocialSecurityNumber(){}
setGender() {}
setDateOfBirth() {}
```

Program Space

```
// Data-attributes
socialSecurityNumber;
gender;
dateOfBirth;

// Behavior-methods
getSocialSecurityNumber() {}
getGender() {}
getDateOfBirth() {}
setSocialSecurityNumber(){}
setGender() {}
setDateOfBirth() {}
```

Reference: Mary

Classes Are Object Templates



Let’s design an UML class for Person

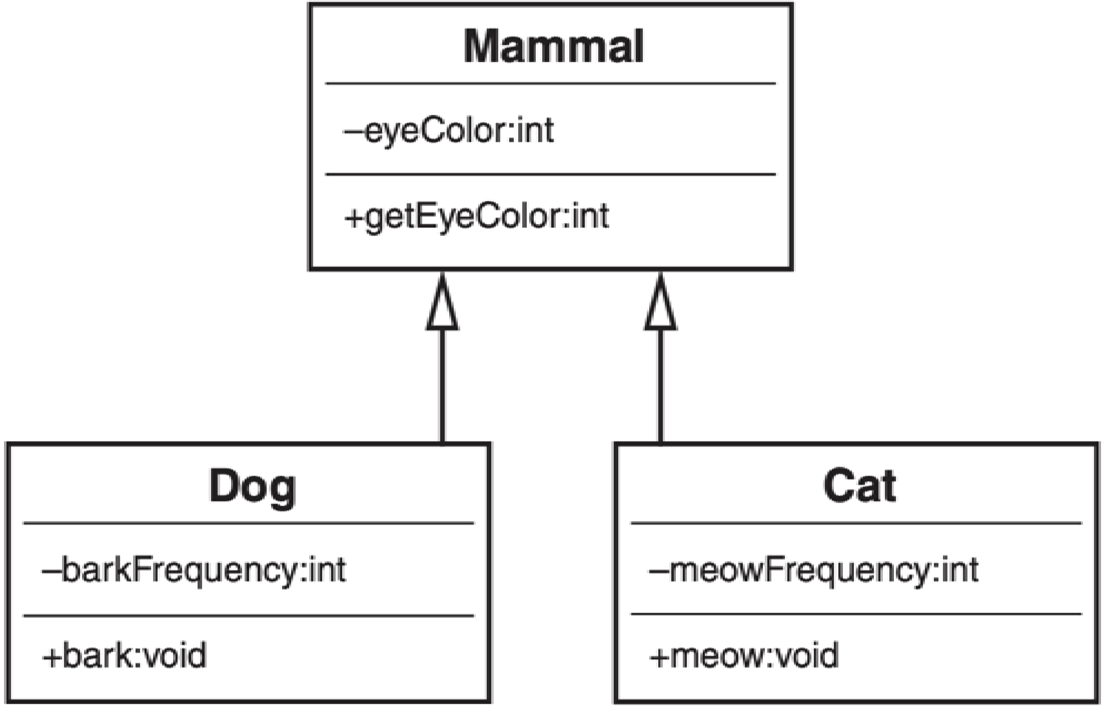
And design an UML class for Student

What is the relationship between
a `Person` class and a `Student` class?

Inheritance

- **For reusing code** – Inheritance allows subclasses to inherit functionality from a superclass, reducing code clone
- **For extending functionality** – With inheritance, subclasses can add or override methods, extending the behavior of the parent class without modifying its code.
- **For establishing relationships** – Inheritance creates a natural hierarchy between classes

Mammal hierarchy



Mammal

–eyeColor:int

+getEyeColor:int

Dog

–barkFrequency:int

+bark:void

Cat

–meowFrequency:int

+meow:void

Superclasses and Subclasses

- **For code reuse** : A **subclass** inherits attributes and methods from its **superclass**,
- **For specialization**: A **subclass** can extend or modify the behavior of the **superclass**
- **For clear structure**: The **superclass** defines general properties and methods, while the **subclass** refines or builds upon them

Abstraction

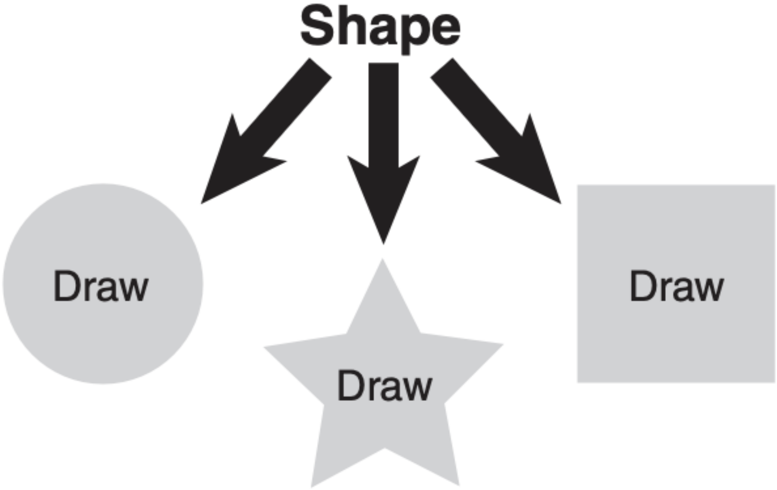
- Abstraction allows developers to hide complex implementation details
- Abstraction allows only expose the essential features
- Abstraction helps separate the interface
- By focusing on "what" an object does rather than "how" it does it

Example of Abstraction

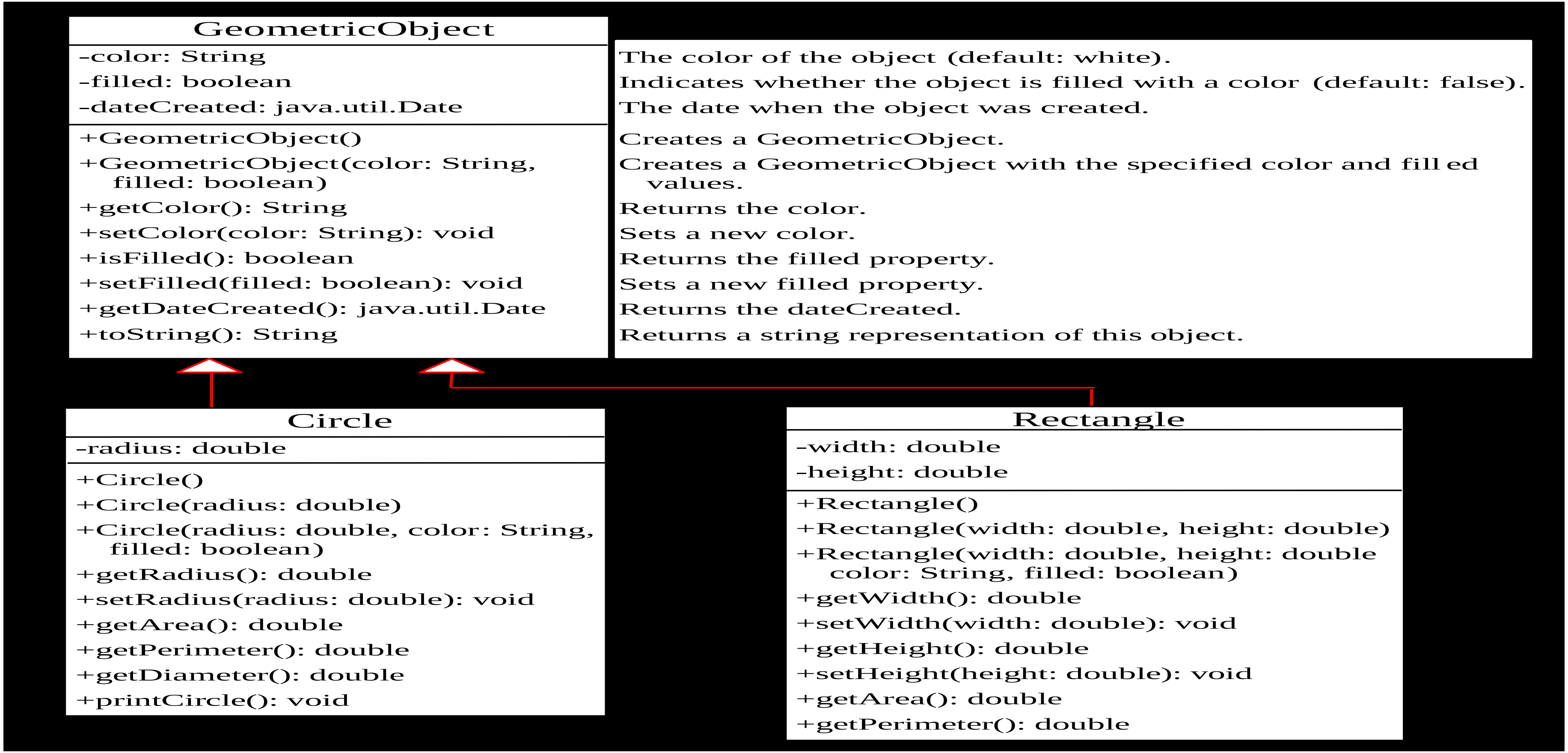
```java
package interface_example;

interface AnimalSound {
   // Abstract method (no implementation)
   void makeSound();
}


class Cat implements AnimalSound {
   // Implementing the abstract method from the interface
   public void makeSound() {
       System.out.println("Meow! Meow!");
   }
}
```

Is-a Relationships

How can an object be an 'is-a' object of other objects?



Super classes and Subclasses



GeometricObject

-color: String
-filled: boolean
-dateCreated: java.util.Date

+GeometricObject()
+GeometricObject(color: String, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String

The color of the object (default: white).
Indicates whether the object is filled with a color (default: false).
The date when the object was created.
Creates a GeometricObject.
Creates a GeometricObject with the specified color and filled values.
Returns the color.
Sets a new color.
Returns the filled property.
Sets a new filled property.
Returns the dateCreated.
Returns a string representation of this object.

Circle

-radius: double

+Circle()
+Circle(radius: double)
+Circle(radius: double, color: String, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getArea(): double
+getPerimeter(): double
+getDiameter(): double
+printCircle(): void

Rectangle

-width: double
-height: double

+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double color: String, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double

Are superclass’s Constructor Inherited?

No. They are not inherited.

They are invoked explicitly (using the *super* keyword) or implicitly.

- A constructor is used to construct an instance of a class.
- Unlike properties and methods, a superclass's constructors are not inherited in the subclass.
- They can only be invoked from the subclasses' constructors, using the keyword *super*.
- If the keyword *super* is not explicitly used, the superclass's no-arg constructor is automatically invoked.

Superclass's Constructor Is Always Invoked

A constructor may invoke an overloaded constructor or its superclass's constructor. If none of them is invoked explicitly, the compiler puts super() as the first statement in the constructor. For example,

```
public A() {
}
```

is equivalent to

```
public A() {
  super();
}
```

```
public A(double d) {
  // some statements
}
```

is equivalent to

```
public A(double d) {
  super();
  // some statements
}
```

Using the Keyword `super`

The keyword `super` refers to the superclass of the class in which `super` appears. This keyword can be used in two ways:

- To call a superclass constructor
- To call a superclass method

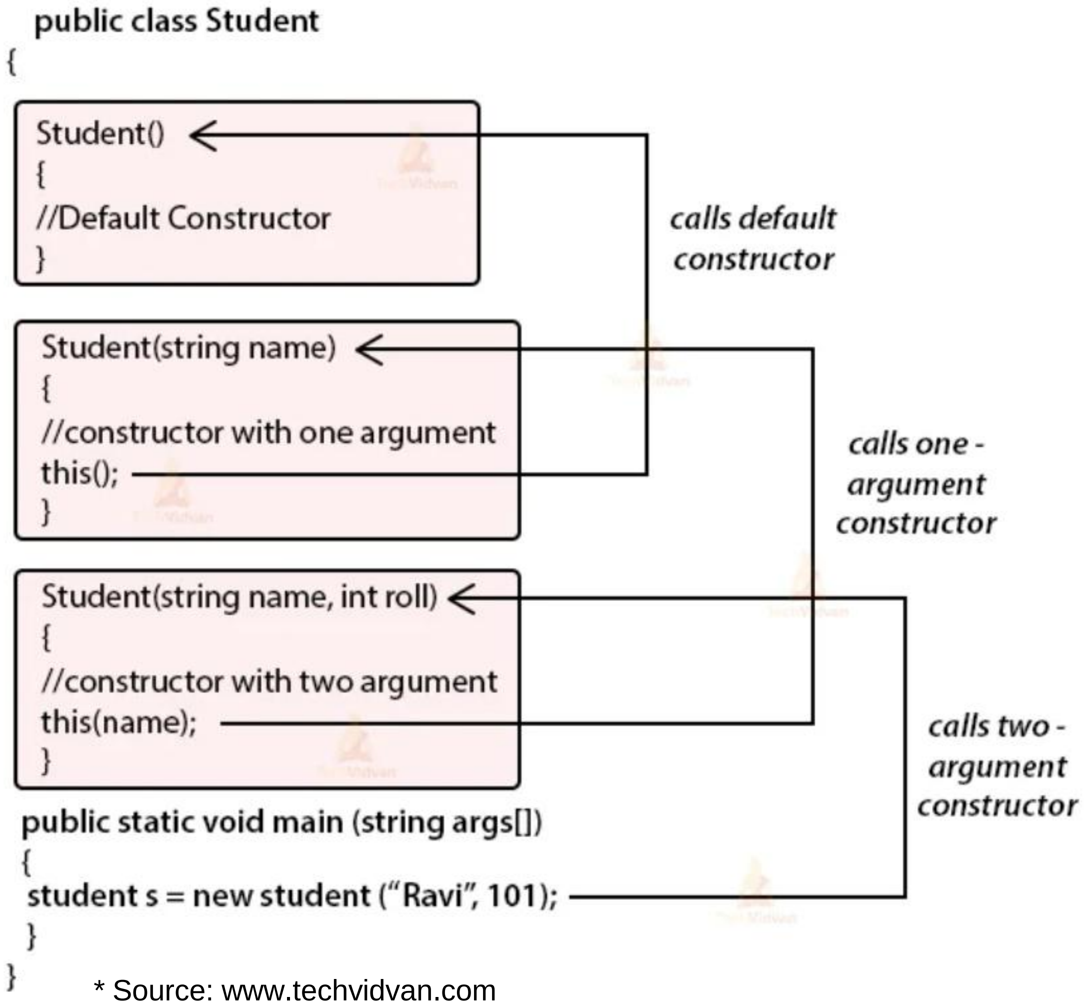
CAUTION

You must use the keyword super to call the superclass constructor. Invoking a superclass constructor’s name in a subclass causes a syntax error.

Java requires that the statement that uses the keyword super appear first in the constructor.

Constructor Chain

Constructing an instance of a class invokes all the superclasses’ constructors along the inheritance chain. This is called *constructor chaining*.



```
public class Student
{
 Student()
 {
 //Default Constructor
 }

 Student(string name)
 {
 //constructor with one argument
 this();
 }

 Student(string name, int roll)
 {
 //constructor with two argument
 this(name);
 }
public static void main (string args[])
{
 student s = new student (“Ravi”, 101);
 }
}
```

calls default constructor
calls one - argument constructor
calls two - argument constructor

* Source: www.techvidvan.com

Constructor Chaining

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }
  public Faculty() {
    System.out.println("Faculty's constructor is invoked");
  }  }

class Employee extends Person {
  public Employee() {
    System.out.println("Employee's constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
  }  }
class Person {
  public Person() {
    System.out.println("Person's constructor is invoked");
  }  }
```

animation

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }
  public Faculty() {
    System.out.println("Faculty's constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    System.out.println("Employee's constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
  }
}
class Person {
  public Person() {
    System.out.println("Person's constructor is invoked");
  }
}
```

1. Start from the main method

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    System.out.println("Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("Person's no-arg constructor is invoked");
  }
```

2. Invoke Faculty constructor

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("Faculty's no-arg construct
  }
}

class Employee extends Person {
  public Employee() {
         System.out.println("Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("Person's no-arg constructor is invoked");
  }
```

3. Invoke Employee’s no-arg constructor

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    System.out.println("Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("Person's no-arg constructor is invoked");
  }
```

4. Invoke Employee’s arg constructor

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    System.out.println("Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("Person's no-arg constructor is invoked");
  }
```

5. Invoke Person() constructor

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
     System.out.println("Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("Person's no-arg constructor is invoked");
  }
```

6. Execute println

Output Sequence?

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("Faculty's no-arg constructor is invoked")
  }
}

class Employee extends Person {
  public Employee() {
    System.out.println("Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("Person's no-arg constructor is invoked");
  }
}
```

```java
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("Faculty's no-arg constructor is invoked")
  }
}

class Employee extends Person {
  public Employee() {
    System.out.println("Employee's no-arg constructor is invoked");
  }

  public Employee(St
    System.out.print
  }
}

class Person {
  public Person() {
    System.out.println("Person's no-arg constructor is invoked");
  }
}
```

Output Sequence?

```
Person's no-arg constructor is invoked
Employee's no-arg constructor is invoked
Faculty's no-arg constructor is invoked
```

Example on the Impact of a Superclass without no-arg Constructor

Find out the errors in the program:

```
public class Apple extends Fruit {
}

class Fruit {
  public Fruit(String name) {
    System.out.println("Fruit's constructor is invoked");
  }
}
```

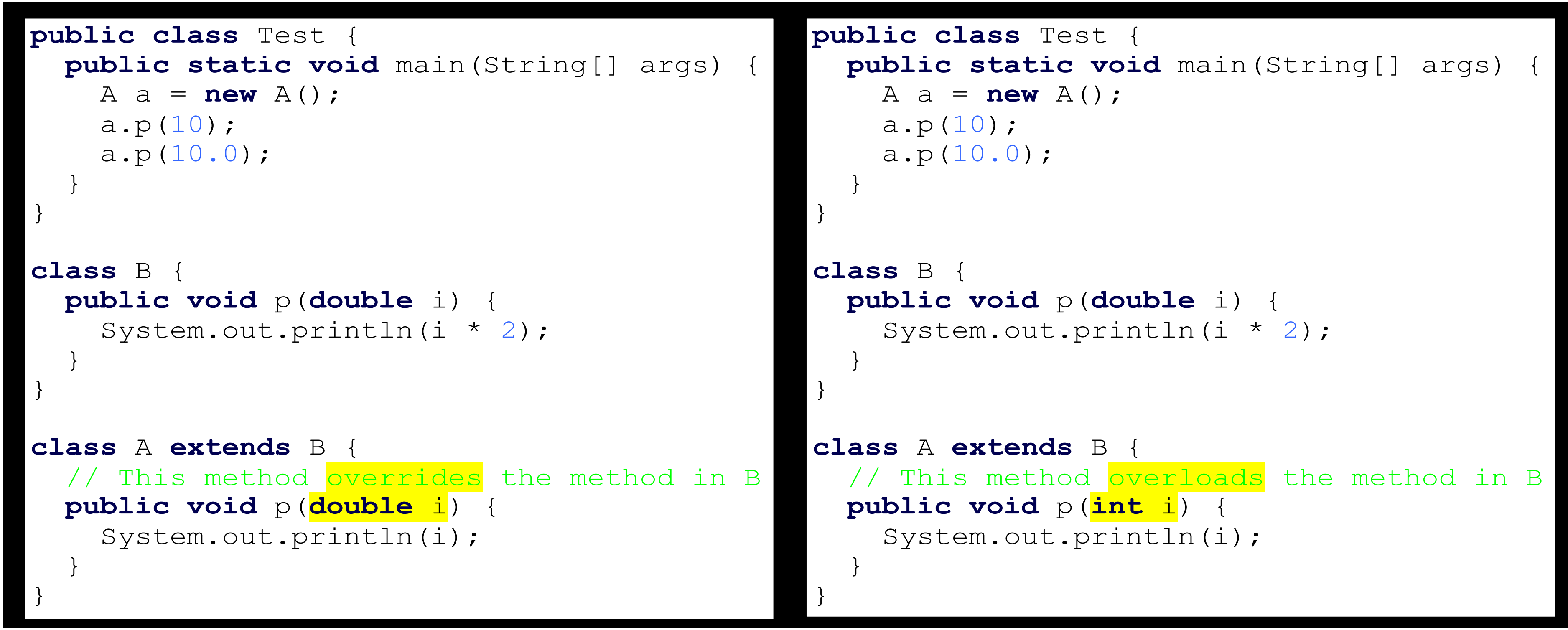
Example on the Impact of a Superclass without no-arg Constructor

If there is no constructor, default no-arg one is added.

If there is a constructor of any kind, the default no-arg constructor is not added.

```
public class Apple extends Fruit {
}

class Fruit {
  public Fruit(String name) {
    System.out.println("Fruit's constructor is invoked");
  }
}
```

Overriding vs. Overloading



```java
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

```java
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

NOTE

An instance method can be overridden only if it is accessible. Thus a private method cannot be overridden, because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

The toString() method in Object

The toString() method returns a string representation of the object. The default implementation returns a string consisting of a class name of which the object is an instance, the at sign (@), and a number representing this object.

```
Loan loan = new Loan();
System.out.println(loan.toString());
```

The code displays something like Loan@15037e5 . This message is not very helpful or informative. Usually you should override the toString method so that it returns a digestible string representation of the object.

Polymorphism

- Polymorphism means *many* (poly) *shapes* (morph)
- In Java, polymorphism refers to the fact that you can have multiple methods with the same name in the same class
- There are two kinds of polymorphism:
- Overloading
  - Two or more methods with different signatures
- Overriding
  - Replacing an inherited method with another having the same signature

Polymorphism

- Polymorphism allows a subclass to provide its specific implementation of a method that is already defined in its superclass
- With polymorphism, a single method or function can operate on different types of objects
- By using polymorphism, we can write cleaner and more concise code, as we don’t need to manually check the object type before performing operations

Overloading

```
class Test {
    public static void main(String args[]) {
        myPrint(5);
        myPrint(5.0);
    }

    static void myPrint(int i) {
        System.out.println("int i = " + i);
    }

    static void myPrint(double d) { // same name, different parameters
        System.out.println("double d = " + d);
    }
}
```

```
int i = 5
double d = 5.0
```

Why overload a method?

DRY (Don’t Repeat Yourself)

When you overload a method with another, very similar method, only one of them should do most of the work:

```
void debug() {
       System.out.println("first=" + first + ", last="+ last);

       ... ...

       System.out.println();
 }

 void debug(String s) {
       System.out.println("At checkpoint " + s + ":");
       debug();
 }
```

Another reason to overload methods

- You may want to do “the same thing” with different kinds of data:

  -
    ```
    class Student extends Person {
       ...
       void printInformation() {
           printPersonalInformation();
           printGrades();
        }
    }
    ```

  -
    ```
    class Professor extends Person() {
        ...
        void printInformation() {
           printPersonalInformation();
           printResearchInterests();
        }
    }
    ```
- Java’s `print` and `println` methods are heavily overloaded

Legal assignments

```
class Test {
    public static void main(String args[]) {
        double d;
        int i;
        d = 5;                          // legal
        i = 3.5;                        // illegal
        i = (int) 3.5;                // legal
    }
}
```

- Widening is legal
- Narrowing is illegal (unless you cast)

Legal method calls

```
class Test {
    public static void main(String args[]) {
        myPrint(5);
    }

    static void myPrint(double d) {
        System.out.println(d);
    }
}
```

```
5.0
```

- Legal because parameter transmission is equivalent to assignment
- myPrint(5) is like double d = 5; System.out.println(d);

Illegal method calls

```
class Test {
    public static void main(String args[]) {
        myPrint(5.0);
    }

    static void myPrint(int i) {
        System.out.println(i);
    }
}
```

myPrint(int) in Test cannot be applied to (double)

- Illegal because parameter transmission is equivalent to assignment
- myPrint(5.0) is like int i = 5.0; System.out.println(i);

Superclass construction

- Unless you specify otherwise, every constructor calls the *default* constructor for its superclass

    -
      ```
      class Foo extends Bar {
          Foo() { // constructor
              super();  // invisible call to superclass constructor
          ...
      ```
- You can use this(...) to call another constructor in the same class:

    -
      ```
      class Foo extends Bar {
          Foo(String message) { // constructor
              this(message, 0, 0); // explicit call to another constructor
          ...
      ```
- You can use super(...) to call a specific *superclass* constructor

    -
      ```
      class Foo extends Bar {
          Foo(String name) { // constructor
              super(name, 5);  // explicit call to superclass constructor
          ...
      ```
- Since the call to another constructor must be the *very first thing you do* in the constructor, you can only do *one* of the above

Summary

- We can think the relation between multiple classes
- Can make an inheritance
- Can design a bit complex scenario with superclass and subclass
- Can write multiple methods with same name
- Can distinguish method overloading and overriding
- Can think about the construction chain
- Can design basic OOP problems

See you next day …