Representing More Complex Forms of Data

Dr. Youna Jung
Northeastern University
yo.jung@northeastern.edu

Object-Oriented Thinking

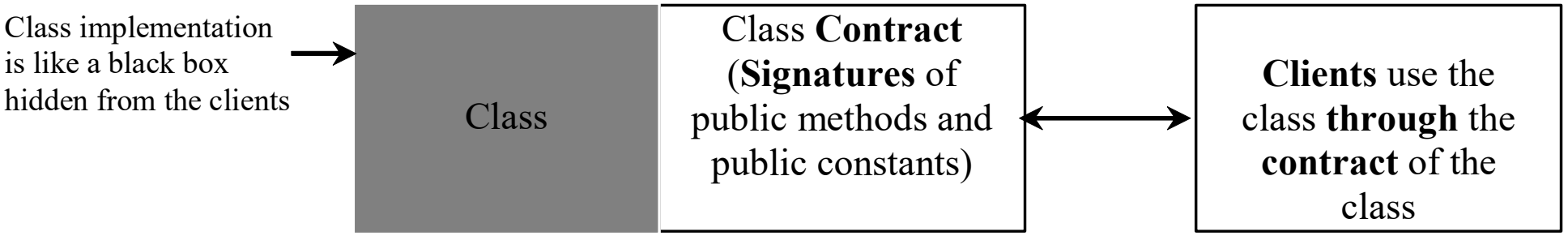
- **Differences** between the procedural programming and object-oriented programming

- Foundation for object-oriented programming
  - ✓ **Classes** ➔ More **Flexibility** and **Modularity** ➔ **Reusable**

Class **Abstraction** and **Encapsulation**

❑ Class abstraction

- ✓ **separate** class **implementation** from the **use** of the class
  - User of the class **does not need to know** how the class is **implemented**.
  - ➔ The detail of implementation is **encapsulated** and hidden from the user.



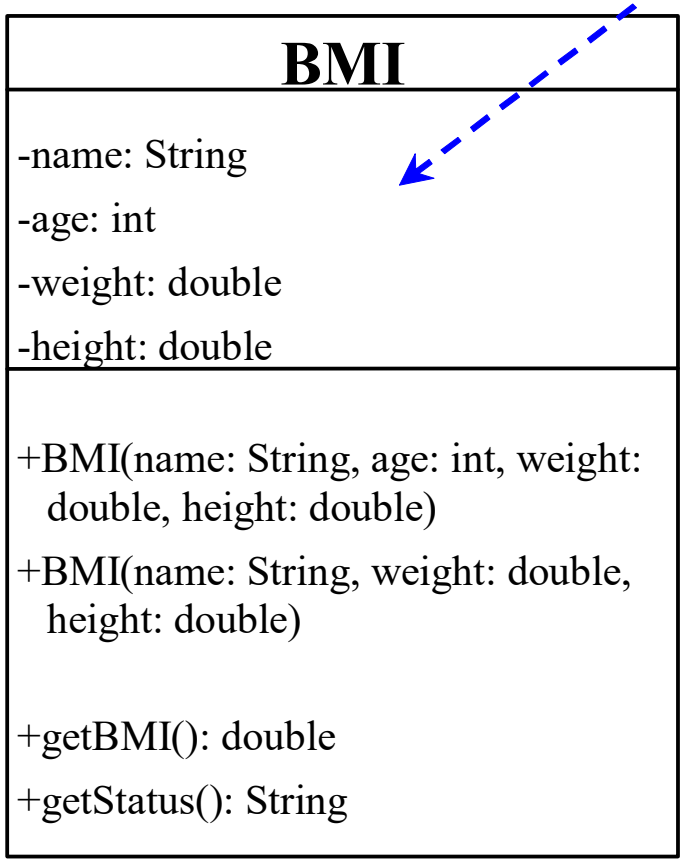
Designing the `Loan` Class

Loan	
-annualInterestRate: double = 2.5	The annual interest rate of the loan (default: 2.5).
-numberOfYears: int = 1	The number of years for the loan (default: 1)
-loanAmount: double = 1000	The loan amount (default: 1000).
-loanDate: Date	The date this loan was created.
+Loan()	Constructs a default Loan object.
+Loan(annualInterestRate: double, numberOfYears: int, loanAmount: double)	Constructs a loan with specified interest rate, years, and loan amount.
+getAnnualInterestRate(): double	Returns the annual interest rate of this loan.
+getNumberOfYears(): int	Returns the number of the years of this loan.
+getLoanAmount(): double	Returns the amount of this loan.
+getLoanDate(): Date	Returns the date of the creation of this loan.
+setAnnualInterestRate(annualInterestRate: double): void	Sets a new annual interest rate to this loan.
+setNumberOfYears(numberOfYears: int): void	Sets a new number of years to this loan.
+setLoanAmount(loanAmount: double): void	Sets a new amount to this loan.
+getMonthlyPayment(): double	Returns the monthly payment of this loan.
+getTotalPayment(): double	Returns the total payment of this loan.

Loan

TestLoanClass

The BMI Class



The get methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

The name of the person.

The age of the person.

The weight of the person in pounds.

The height of the person in inches.

Creates a BMI object with the specified name, age, weight, and height.

Creates a BMI object with the specified name, weight, height, and a **default age 20**.

Returns the BMI

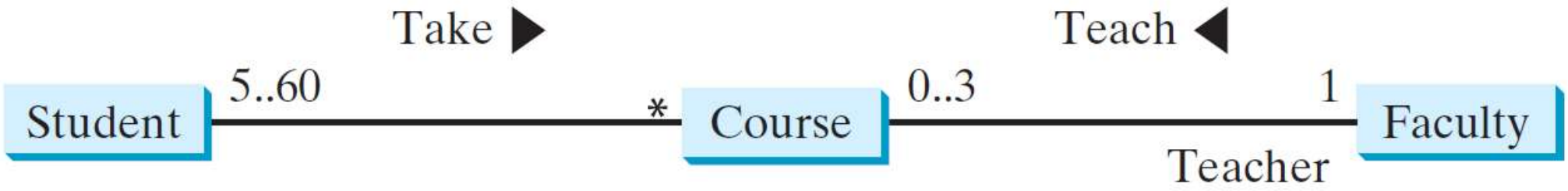
Returns the BMI status (e.g., normal, overweight, etc.)

BMI

UseBMIClass

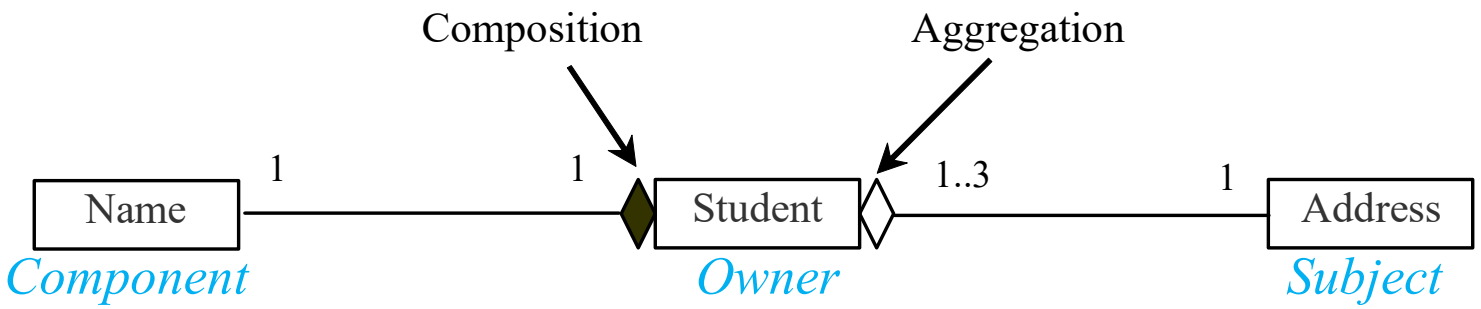
Class Relationships

- **Association**
  - ✓ A **general binary relationship** that describes an activity between two classes.
- **Composition**
- **Aggregation**
- **Inheritance**



Aggregation VS Composition

- **Aggregation** (*has-a* relationships)
  - ✓ represents an **ownership** relationship between two objects
    - The **owner** class/object
      - ***Aggregating** object* and ***Aggregating** class*
    - The **subject** class/object
      - ***Aggregated** object* and its class an ***Aggregated** class*.
- **Composition**
  - ✓ A **special case** of the **aggregation** relationship
    - If the **owner cannot exist without subject**



Aggregation

- An aggregation relationship is usually represented as a **data field** in the **owner class**

```
public class Name {
   ...
}
```

Aggregated class

Subject

```
public class Student {
   private Name name;
   private Address address;

   ...
}
```

Aggregating class

Owner

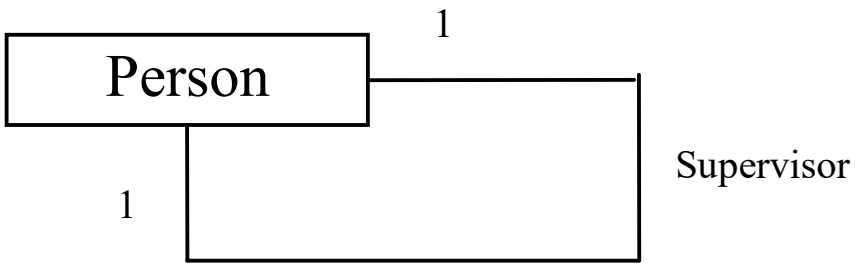
```
public class Address {
   ...
}
```

Aggregated class

Subject

Aggregation Between Same Class

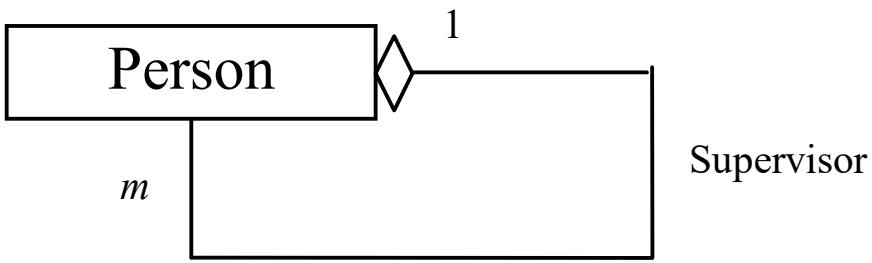
- Aggregation may exist between objects of the same class
  - ✓ E.g.) A person may have a supervisor.




```
public class Person {
  // The type for the data is the class itself
  private Person supervisor;
  ...
}
```

Aggregation Between Same Class

What happens if a person has several supervisors?




```
public class Person {
   ...
   private Person[] supervisors;
}
```

Example: The `Course` Class

Course
-courseName: String -students: String[] -numberOfStudents:int = 0
+Course(courseName: String) +getCourseName(): String +addStudent(student: String): void +dropStudent(student: String): void +getStudents(): String[] +getNumberOfStudents(): int

The name of the course.

An array to store the students for the course.

The number of students (default: 0).

Creates a course with the specified name.

Returns the course name.

Adds a new student to the course.

Drops a student from the course.

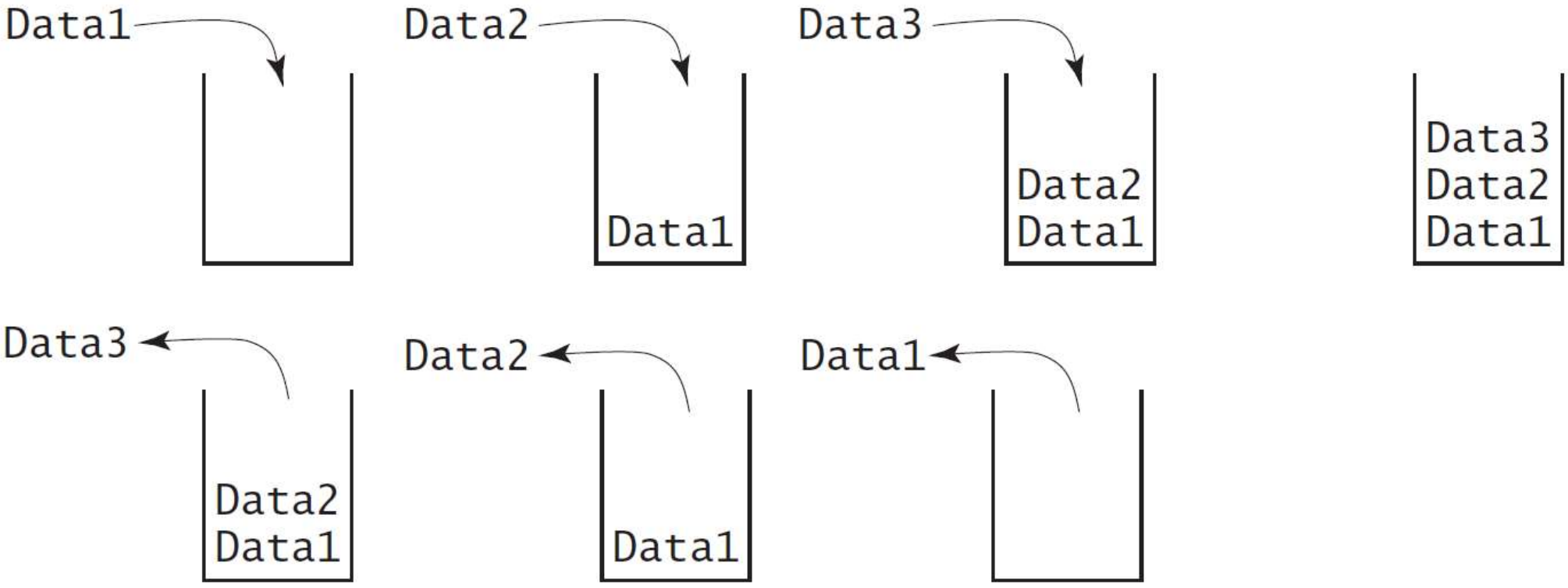
Returns the students in the course.

Returns the number of students in the course.

Course

TestCourse

Designing the `StackOfIntegers` Class

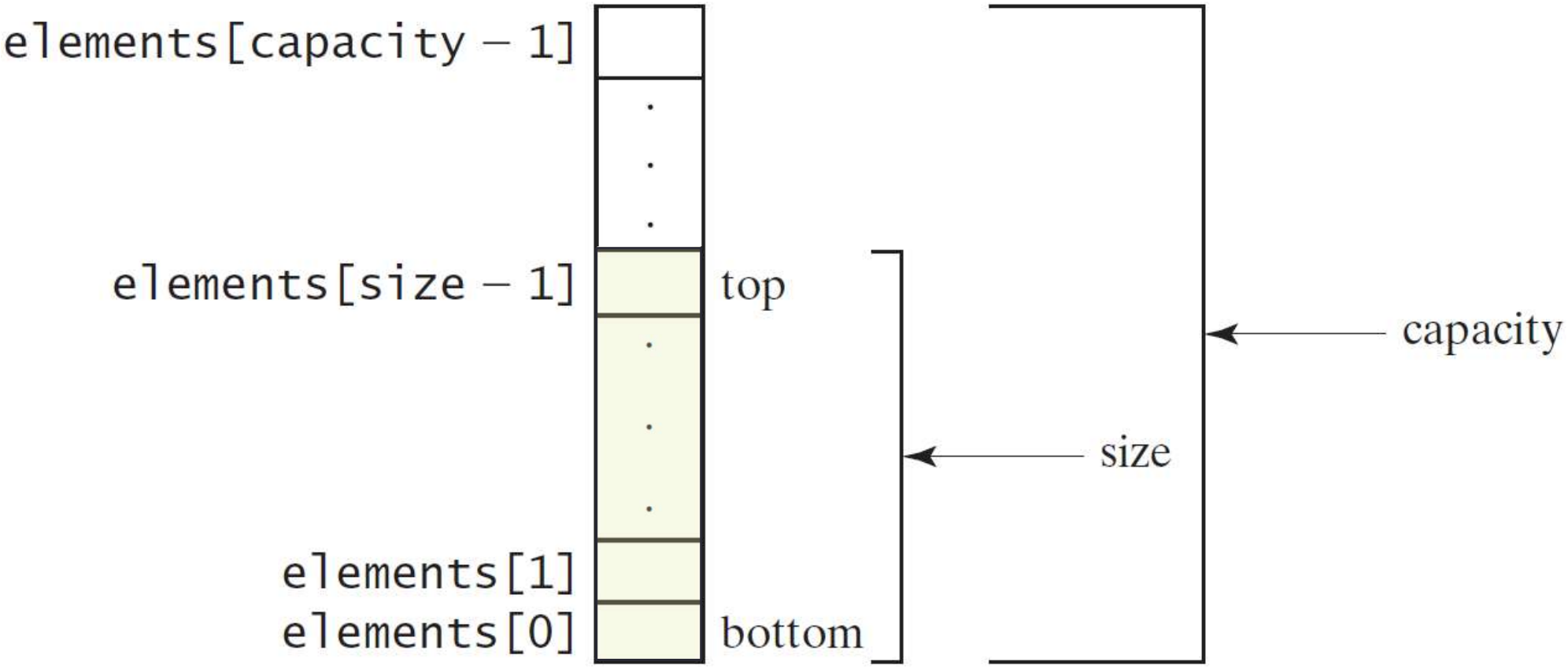


The `StackOfIntegers` Class

StackOfIntegers	
-elements: int[]	An array to store integers in the stack.
-size: int	The number of integers in the stack.
+StackOfIntegers()	Constructs an empty stack with a default capacity of 16.
+StackOfIntegers(capacity: int)	Constructs an empty stack with a specified capacity.
+empty(): boolean	Returns true if the stack is empty.
+peek(): int	Returns the integer at the top of the stack without removing it from the stack.
+push(value: int): int	Stores an integer into the top of the stack.
+pop(): int	Removes the integer at the top of the stack and returns it.
+getSize(): int	Returns the number of elements in the stack.

TestStackOfIntegers

Implementing `StackOfIntegers` Class



StackOfIntegers

Wrapper Classes

- Provide a **way** to **use primitive data types** as **objects**
  - ✓ The wrapper classes do not have ~~no-arg constructors~~.
  - ✓ The instances of all wrapper classes are **immutable**

Primitive Data Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

The Integer and Double Classes

java.lang.Integer
-value: int
+MAX_VALUE: int
+MIN_VALUE: int
+Integer(value: int)
+Integer(s: String)
+byteValue(): **byte**
+shortValue(): **short**
+intValue(): **int**
+longVlaue(): **long**
+floatValue(): **float**
+doubleValue():**double**
+compareTo(o: Integer): int
+toString(): **String**
+valueOf(s: String): **Integer**
+valueOf(s: String, radix: int): **Integer**
+parseInt(s: String): **int**
+parseInt(s: String, radix: int): **int**

java.lang.Double
-value: double
+MAX_VALUE: double
+MIN_VALUE: double
+Double(value: double)
+Double(s: String)
+byteValue(): **byte**
+shortValue(): **short**
+intValue(): **int**
+longVlaue(): **long**
+floatValue(): **float**
+doubleValue():**double**
+compareTo(o: Double): int
+toString(): **String**
+valueOf(s: String): **Double**
+valueOf(s: String, radix: int): **Double**
+parseDouble(s: String): **double**
+parseDouble(s: String, radix: int): **double**

Conversion Methods

- Each **numeric wrapper** class implements the **conversion methods** defined in the **`Number`** class
  - ✓ convert **objects** into **primitive type** values
    - – **`doubleValue()`**, **`floatValue()`**, **`intValue()`**, **`longValue()`**, and **`shortValue()`**

17

Static valueOf(s)

- The numeric wrapper classes have **valueOf(String s)**
  - ✓ **creates** a new **object initialized** to the **value** represented by the specified number string.

```
Double doubleObject = Double.valueOf("12.4");

Integer integerObject = Integer.valueOf("12");
```


18

Automatic Conversion

- JDK 1.5 allows primitive type and wrapper classes to be **converted automatically**.

```
Integer[] intArray = {new Integer(2),
  new Integer(4), new Integer(3)};
```

(a)

Equivalent

```
Integer[] intArray = {2, 4, 3};
```

(b)

New JDK 1.5 **boxing**
(Primitive type → Object)

```
Integer[] intArray = {2, 4, 3};
System.out.println(intArray[0] + intArray[1] + intArray[2]);
```

Unboxing
(Object→ Primitive type)

INHERITANCE

Inheritance

- Suppose you will define classes to model **circles**, **rectangles**, and **triangles**.
  - ✓ These classes have many **common features** (*e.g. they can be drawn in a certain color and be filled or unfilled*). What is the best way to design these classes so to **avoid redundancy**?

use **Inheritance**!

- enables you to define a general class (***superclass***) and later extend it to more specialized classes (***subclasses***)

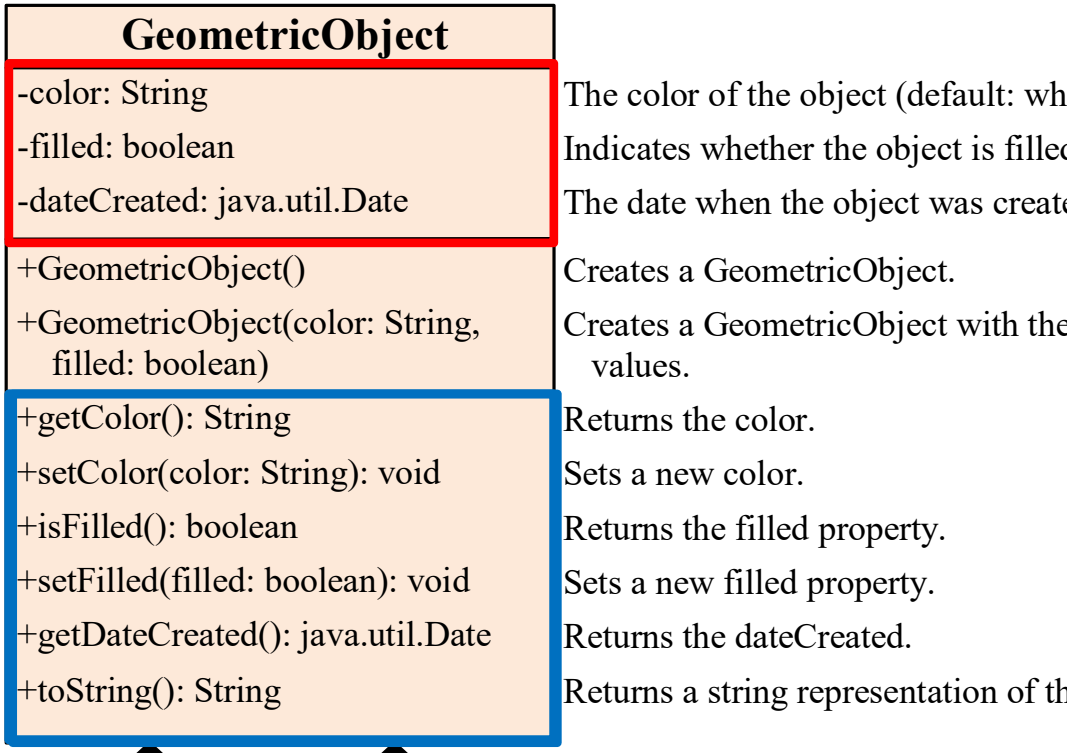
Inheritance

- The way to **define new classes** from **existing** classes (reusing software)
  - ✓ used to model the ***is-a*** relationship
  - ✓ Java does not allow ~~***multiple inheritance***~~
- A class `C1` extended from another class `C2`.
  - ✓ `C2` is called a ***superclass*** (*parent* or *base* class)
  - ✓ `C1` is called a ***subclass*** (*child/extended/derived* class)
    - **inherits** accessible **data fields** and **methods** **from** its superclass (inheritance)
      - ➢ Only accessible members
        - → `private` members cannot be inherited!
        - → can be accessed through `public` **accessor** or **mutator**
    - AND may also **add** new data fields and methods (**extension/specialization**)

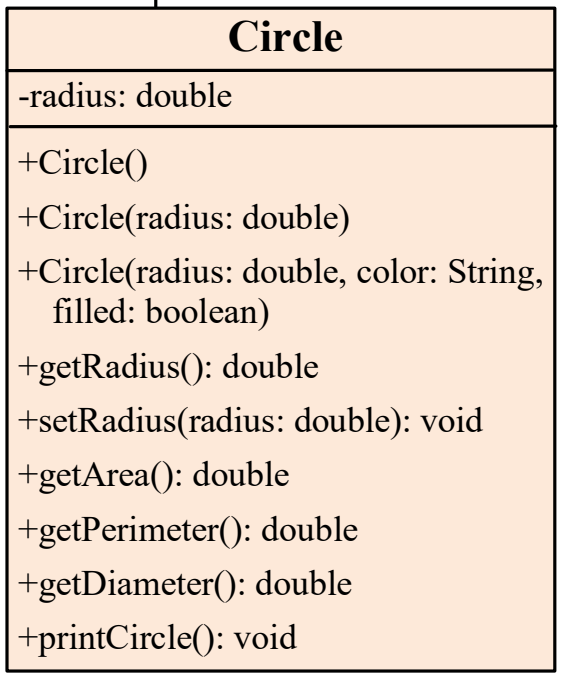
superclass

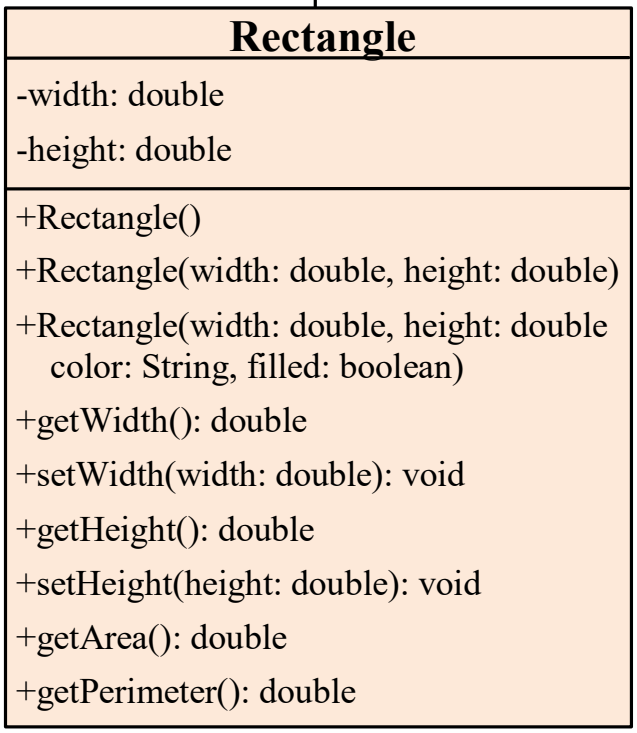
`Private` *member* → *Cannot be inherited*

`Public` *member* → *Inherited to subclass*



subclass



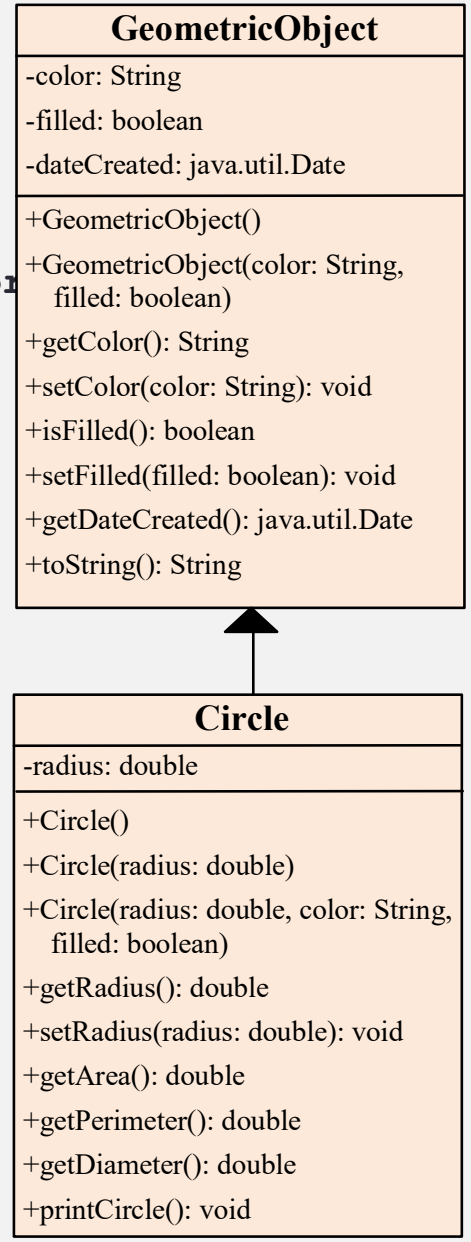


subclass

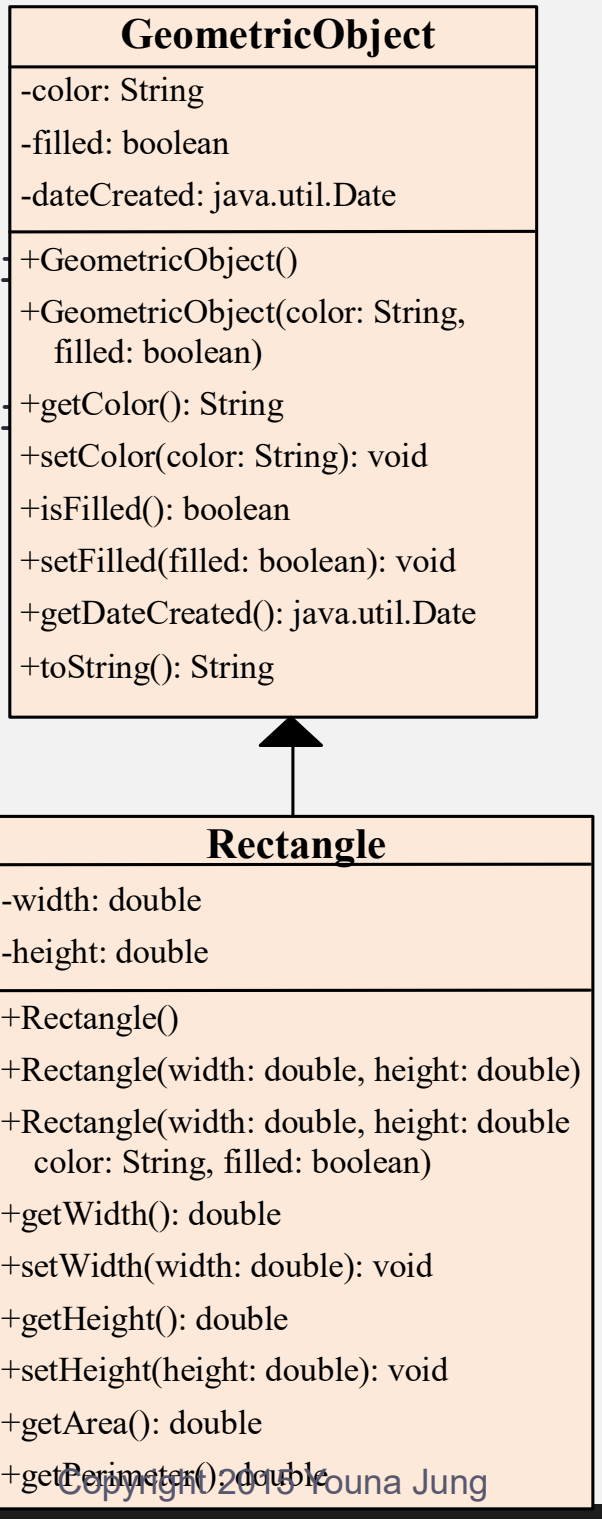
```
public class SimpleGeometricObject {
  private String color = "white";
  private boolean filled;
  private java.util.Date dateCreated;

public SimpleGeometricObject() {
    dateCreated = new java.util.Date();   }
public SimpleGeometricObject(String color, boolean filled) {
    dateCreated = new java.util.Date();
    this.color = color;
    this.filled = filled;  }

  public String getColor() {
    return color;  }

  public void setColor(String color) {
    this.color = color;  }

  public boolean isFilled() {
    return filled;  }

  public void setFilled(boolean filled) {
    this.filled = filled;  }

  public java.util.Date getDateCreated() {
    return dateCreated;  }

  public String toString() {
    return "created on " + dateCreated + "\ncolor: " + color +
      " and filled: " + filled;  }   }
```

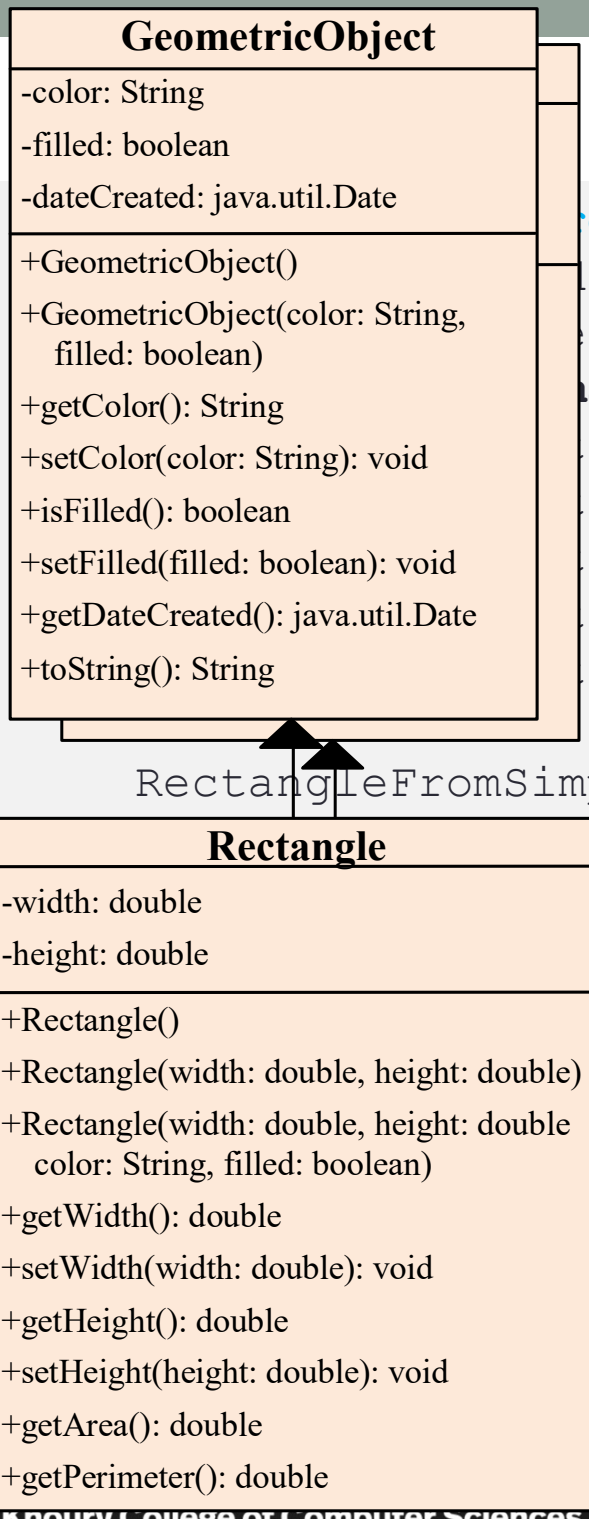
```
public class CircleFromSimpleGeometricObject extends SimpleGeometricObject {
  private double radius;

public CircleFromSimpleGeometricObject() {
}
public CircleFromSimpleGeometricObject(double radius) {
    this.radius = radius;  }
public CircleFromSimpleGeometricObject(double radius, String color
    this.radius = radius;
    setColor(color);
    setFilled(filled);  }

  public double getRadius() {
    return radius;  }
  public void setRadius(double radius) {
    this.radius = radius;  }

  public double getArea() {
    return radius * radius * Math.PI;  }

  public double getDiameter() {
    return 2 * radius;  }

  public double getPerimeter() {
    return 2 * radius * Math.PI;  }

  public void printCircle() {
    System.out.println("The circle is created " + getDateCreated() +
      " and the radius is " + radius);  } }
```

```
this.color = color;    // Error!
this.filled = filled;  // Error!
```



```
public class RectangleFromSimpleGeometricObject extends SimpleGeometricObject {
  private double width;
  private double height;

public RectangleFromSimpleGeometricObject() {
}
public RectangleFromSimpleGeometricObject(double width, doubl
    this.width = width;
    this.height = height; }
public RectangleFromSimpleGeometricObject(double width, doubl
        String color, boolean filled) {
    this.width = width;
    this.height = height;
    setColor(color);
    setFilled(filled);  }

  public double getWidth() {
    return width;  }
  public void setWidth(double width) {
    this.width = width; }

  public double getHeight() {
    return height;  }
  public void setHeight(double height) {
    this.height = height;  }

  public double getArea() {
    return width * height;  }

  public double getPerimeter() {
    return 2 * (width + height);  }}
```






```
cleRectangle {
 main(String[] args) {
GeometricObject circle =
SimpleGeometricObject(1);
ln("A circle " + circle.toString());
ln("The color is " + circle.getColor());
ln("The radius is " + circle.getRadius());
ln("The area is " + circle.getArea());
ln("The diameter is " + circle.getDiameter());

RectangleFromSimpleGeometricObject rectangle =
romSimpleGeometricObject(2, 4);
ln("\nA rectangle " + rectangle.toString());
ln("The area is " + rectangle.getArea());
ln("The perimeter is " + rectangle.getPerimeter());
```

the Keyword `super`

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

1) To call a **superclass constructor**
2) To call a **superclass method**

Are Superclass's Constructor Inherited?

- **No**! A **superclass's constructors** are **not inherited** in the subclass.
  - ✓ BUT they can be **invoked** explicitly or implicitly.
    - Explicitly using the **`super`** keyword.

```
super()   // invokes the no-arg constructor of its superclass
super(parameters)   // invokes the superclass constructor matched
```

      - Caution: `super()` or `super(para)` **must** be the **first** statement of the **subclass's constructor** !!
      - Caution: ~~**Invoking a superclass constructor by method name** in~~ a subclass causes a **syntax error** !!
    - If the keyword **`super`** is not explicitly used → the **superclass's *no-arg constructor*** is **automatically invoked**.

Constructor Chaining

- When constructing an **object** of a **subclass**, the subclass constructor **first** invokes its **superclass constructor before** performing **its own** tasks
  - ✓ ➔ In any case, a constructor invokes the constructors of all the superclasses along the inheritance chain (***constructor chaining***)

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

is equivalent to

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

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

is equivalent to

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

```
public class Faculty extends E
  public static void main(Stri
    new Faculty();  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");  }

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

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");  }
}
```

```
(1) Person's no-arg constructor is invoked
(2) Invoke Employee's overloaded constructor
(3) Employee's no-arg constructor is invoked
(4) Faculty's no-arg constructor is invoked
```

```
Faculty() {
Performs Faculty's
task;
}
```

```
Employee() {
  this("(2)..")
  Performs Employee's
     task;
}
```

```
Employee(String s){
Performs Employee's
   task;
}
```

```
Person() {

Performs person's
task;
}
```

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

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

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

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg 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("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

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

class Person {
  public Person() {
    System.out.println("(1) 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("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

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

class Person {
  public Person() {
    System.out.println("(1) 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("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

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

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

4. Invoke Employee(String) constructor

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

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

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

class Person {
  public Person() {
    System.out.println("(1) 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("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

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

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

6. Execute println

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

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

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

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

7. Execute println

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

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

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

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

8. Execute println

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

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

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

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

9. Execute println

Superclass without no-arg Constructor

```
public class Apple extends Fruit {
}

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

- No constructor is explicitly defined in **`Apple`** → **`Apple`**'s default no-arg constructor is defined implicitly
  - Since `Apple` is a subclass of **`Fruit`**, `Apple`'s default constructor automatically invokes **`Fruit`**'s **no-arg constructor**
- But **`Fruit`** does not have a no-arg constructor

→ **Compile Error!!**

Defining a Subclass

- A subclass **inherits accessible data fields** and **methods** from a **superclass**. In addition, you can also
  - ✓ Add **new data** fields
  - ✓ Add **new methods**
  - ✓ **Override** the methods of the superclass

Calling **Superclass Methods**

- You could **write** the `printCircle()` method in the `Circle` class using the method of its super class as follows:

```
public void printCircle() {
  System.out.println("The circle is created " +
    super.getDateCreated() + " and the radius is " + radius);
}
```

Method Overriding

- Sometimes it is necessary for the **subclass** to **modify** the implementation of a **method** defined in the **superclass**.
  - ✓ To override a method, the **method** must be defined in the **subclass** **using** **the same signature** **and** **the same return type** as in its **superclass**.

```
public class Circle extends GeometricObject {

  // Other methods are omitted

  /** Override the toString method defined in GeometricObject */
  public String toString() {
    return super.toString() + "\nradius is " + radius;  }

}
```

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.

NOTE

- Like an instance method, a `static` method can be **inherited**.

- However, a `static` method **cannot be overridden**.
  - ✓ If a `static` method defined in the superclass is **redefined** in a **subclass**, the method defined in the **superclass** is **hidden**.

Overriding vs. Overloading

- **Overloading**
  - ✓ means to define **multiple methods** with the ***same name*** but ***different signature***.
  - ✓ Overloaded methods can be **either in** the ***same class*** or ***different classes*** related by inheritance

- **Overriding**
  - ✓ have the ***same signature*** *and* the ***same return type***
  - ✓ means to provide a **new implementation** for a method in the **subclass**.
  - ✓ Overridden methods **must** be **in** ***different classes*** related by **inheritance**.
    - Overridden method in a **superclass**
    - Overriding method in a **subclass**

Overriding vs. Overloading

```
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);
  }
}
```

```
10.0
10.0
```

```
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);
  }
}
```

```
10
20.0
```

@Override

- Override annotation (`@Override`)
  - ✓ a special annotation, denotes that the annotated method is required to **override** a method **in** the **superclass**
    - If a method with `@Override` does not override its superclass's method → **Compile Error**!
    - Without `@Override`, cannot catch a mistake.

`Object` Class

- Every class in Java is descended from the `java.lang.Object` class.
  - ✓ If no inheritance is specified when a class is defined, the superclass of the class is `Object`.

```
public class Circle {
   ...
}
```

Equivalent

```
public class Circle extends Object {
   ...
}
```

Method	Behavior
`boolean equals(Object obj)`	Compares this object to its argument.
`int hashCode()`	Returns an integer hash code value for this object.
`String toString()`	Returns a string that textually represents the object.
`Class<?> getClass()`	Returns a unique object that identifies the class of this object.

`toString()` in `Object` class

- returns a **string representation** of the **object**.
  - ✓ The default implementation returns a string consisting of 1) a **class name** of which the object is an instance, 2) the at sign (@), and 3) the **object's memory address** in hexadecimal.

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

```
Loan@15037e5
```

    - – Usually you should override the `toString()` method so that it returns a digestible string representation of the object.

Polymorphism

- A class defines a **type**.
  - ✓ ***Subtype*** : A type defined by a subclass
  - ✓ ***Supertype***: A type defined by its superclass
    - e.g.) **Circle** is a subtype of **GeometricObject**
    - e.g.) **GeometricObject** is a supertype for **Circle**

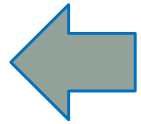
- **Polymorphism** means that a **variable** of a **supertype** can **refer** to a **subtype** object.

Polymorphism

- A **subclass** is a **specialization** of its **superclass**
  - ✓ Every instance of a **subclass** is **also** an **instance** of its **superclass**, but **NOT** vice versa.
    - e.g.) every **`circle`** is a **`geometric`** object,
    - e.g.) Not every `geometric` object is a `circle`
  - ✓ An object of a **subclass** can be **used wherever** its superclass object is used (**Polymorphism***)

* *Polymorphism is from a Greek word meaning "**many forms**"*

PolymorphismDemo.java



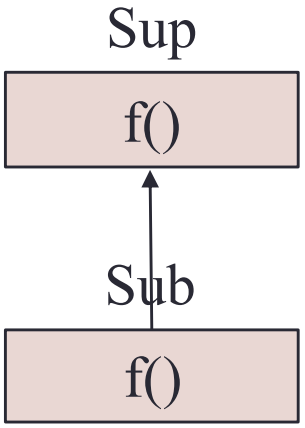
```
public class PolymorphismDemo {
  /** Main method */
  public static void main(String[] args) {
    // Display circle and rectangle properties
    displayObject(new CircleFromSimpleGeometricObject (1, "red", false));
    displayObject(new RectangleFromSimpleGeometricObject(1,1, "black", true));
  }

  /** Display geometric object properties */
  public static void displayObject(SimpleGeometricObject object) {
    System.out.println("Created on " + object.getDateCreated() +
      ". Color is " + object.getColor());
  }
}
```

Created on Thu Mar 14 10:42:23 EDT 2022. Color is red

Created on Thu Mar 14 10:42:24 EDT 2022. Color is black

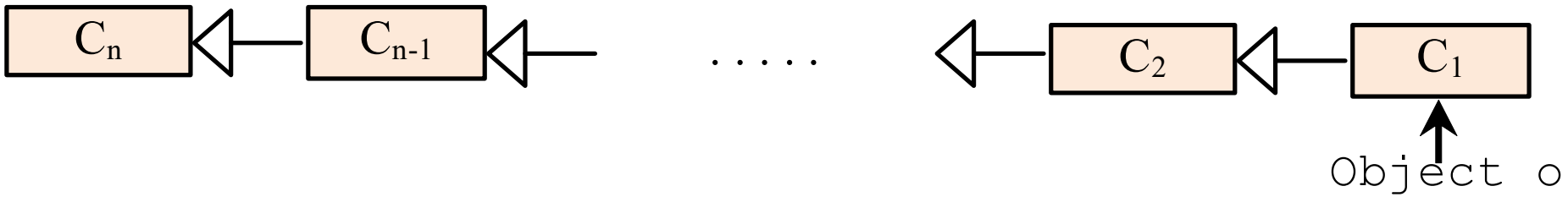
Dynamic Binding

- A **method** can be **implemented** in **several classes** along the inheritance chain.
  - ✓ A method can be defined in a superclass and **overridden** in its subclass
  - ✓ **At runtime**, the **JVM decides which method** is **invoked** (***Dynamic Binding***).



Dynamic Binding Procedure

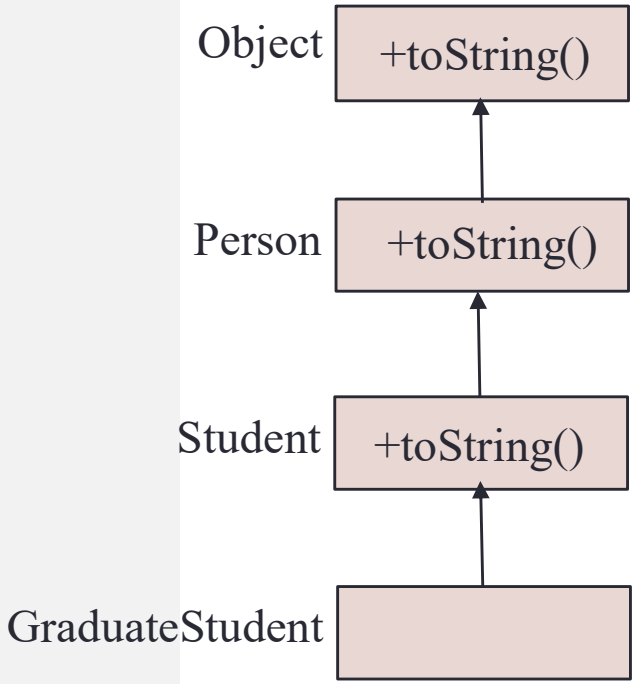
- Suppose an object **o** is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n.
  - ✓ **C_n** is the most general class, and **C_1** is the most specific class.
    - In Java, C_n is the `Object` class.
  - ✓ If **o** invokes a method **`p()`**, the JVM searches the implementation for the method `p()` in C_1, C_2, ..., C_{n-1} and C_n, in this order, until it is found (*The **most specific** takes **precedence***).
  - ✓ Once an implementation is found, the search stops and the first-found implementation is invoked.



Since o is an instance of C_1, o is also an instance of C_2, C_3, and C_n

Polymorphism and Dynamic Binding

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());    }

  public static void m(Object x) {
    System.out.println(x.toString());   }
}

class GraduateStudent extends Student {
}
class Student extends Person {
  public String toString() {
    return "Student";   }}
class Person extends Object {
  public String toString() {
    return "Person";   }
}
```



Student
Student
Person
Java.lang.Object@24f87b

Generic Programming

- Polymorphism allows methods to be used generically for a wide range of object arguments (**Generic programming**).
  - ✓ If a method's parameter type is a superclass (e.g., `Object`), you may pass an object to this method of any of the parameter's subclasses (e.g., `Student` or `String`). (**Polymorphism**)
  - ✓ When an object (e.g., a `Student` object or a `String` object) is used in the method, the particular implementation of the method of the object that is invoked (e.g., `toString()`) is determined dynamically (**Dynamic binding**).

Declared type VS Actual type

- A variable must be declared a type
  - ✓ **declared type**: The type that declares a variable
  - ✓ **actual type**: The actual class for the object referenced by the variable
  - ✓ For example,

```
Object o = new GeometricObject();
System.out.println(o.toString());
```

invoke `toString()` of the `GeometricObject` class

  - – o's declared type is `Object`
  - – o's actual type is `GeometricObject`

Method Matching VS Binding

- Matching a method signature and binding a method implementation are two different issues.
  - ✓ **Method matching**
    - The **compiler** finds a matching method according to parameter type, number of parameters, and order of the parameters at **compile time**.
    - The **declared type** of the reference variable decides a matching method
  - ✓ **Binding**
    - A method may be implemented in several subclasses. The **JVM** dynamically binds the implementation of the method at **runtime**.
    - The **actual type** of the variable decides a method to be invoked

Casting Objects

- You have already used the casting operator to convert variables of one primitive type to another. **Casting** can also be used to **convert** an **object** of **one class type** to **another** within an **inheritance hierarchy**.
  - ✓ **Implicit casting**

```
m(new Student());
```
=
```
Object o = new Student();// Implicit casting
m(o);
```

    - `Object o = new Student()` is legal because an instance of `Student` is automatically an instance of `Object`.
  - ✓ **Explicit casting**

An Object is not necessarily an instance of `Student`. Even though you can see that `o` is really a `Student` object, the compiler is not so clever

```
Student b = o;                  // Error!
```

```
Student b = (Student)o; // Explicit casting
```

Casting from **Superclass** to **Subclass**

- **Explicit casting must** be used when casting an object from a **superclass** to a **subclass**.
  - ✓ Ex) An **apple** is a **fruit**
    - so you can always safely assign an instance of `Apple` to a variable for `Fruit`.
    - However, a fruit is not necessarily an apple, so you have to use **explicit casting** to assign an instance of `Fruit` to a variable of `Apple`.

```
Apple x = (Apple)fruit;

Orange x = (Orange)fruit;
```

The `instanceof` Operator

- Use the `instanceof` operator to **test whether** an **object** is **an instance of a class**
  - ✓ To ensure that the source object is an instance of the target class before performing a casting

```
Object myObject = new Circle();
...

/** Perform casting if myObject is an instance of Circle */
if (myObject instanceof Circle) {
    System.out.println("The circle diameter is " +
      ((Circle)myObject).getDiameter());
    ...
}
```

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

    Object object1 = new CircleFromSimpleGeometricObject(1);
    Object object2 = new RectangleFromSimpleGeometricObject(1, 1);

    displayObject(object1);
    displayObject(object2);
  }

  public static void displayObject(Object object) {
    if (object instanceof CircleFromSimpleGeometricObject) {
      System.out.println("The circle area is " +
        ((CircleFromSimpleGeometricObject)object).getArea());
      System.out.println("The circle diameter is " +
        ((CircleFromSimpleGeometricObject)object).getDiameter());
    }
    else if (object instanceof RectangleFromSimpleGeometricObject) {
      System.out.println("The rectangle area is " +
        ((RectangleFromSimpleGeometricObject)object).getArea());
    }
  }
}
```

The `equals()` Method

- The **`equals()`** method compares the **contents** of two **objects**.
  - ✓ The default implementation of the `equals()` method in the **`Object`** class is as follows:

```
public boolean equals(Object obj) {
      return this == obj;
}
```

  - ✓ The equals() method is overridden in the **`Circle`** class.

```
public boolean equals(Object o) {
  if (o instanceof Circle) {
    return radius == ((Circle)o).radius;}
  else
    return false;}
```

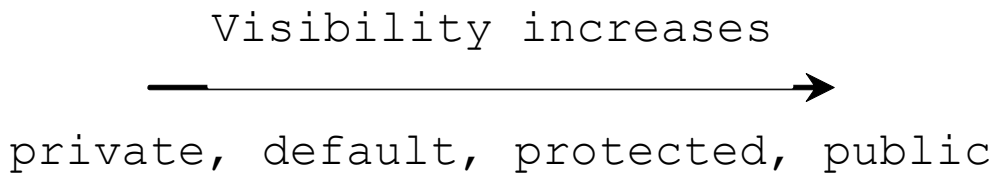
== vs equals()

- The == comparison operator
  - ✓ for comparing two **primitive data type values** OR
  - ✓ for determining whether two objects have the **same references**.

- equals()
  - ✓ test whether two objects have the same **contents**, provided that the method is modified in the defining class of the objects.

`protected` Modifier with subclass

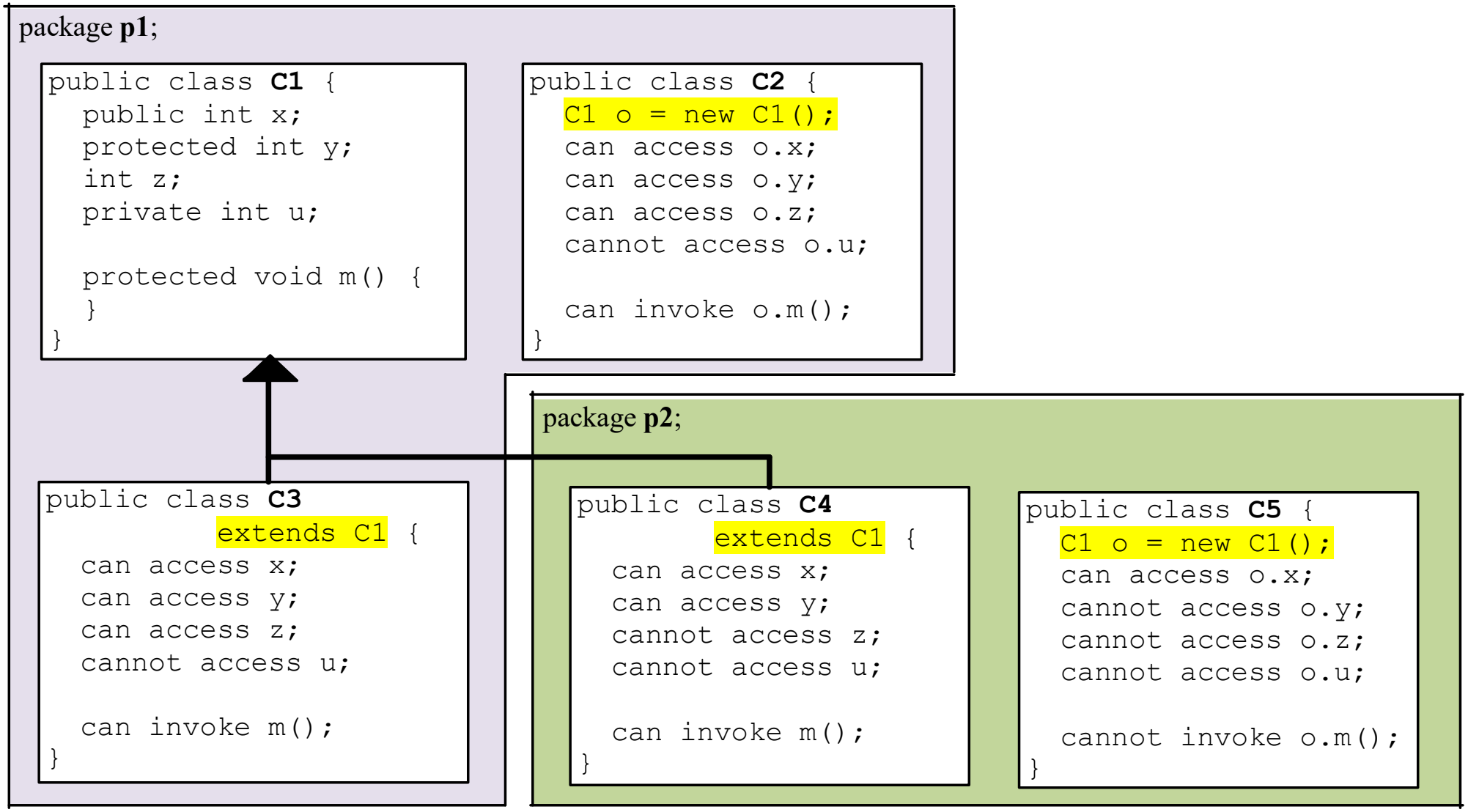
- A **`protected`** data or a **`protected`** method can be accessed by any class in the **same package** or its **subclasses**, even if the **subclasses** are in a **different package**.
  - ✓ The **default** modifier can be accessed by any class in the **same package only**
  - ✓ The `protected` modifier can be applied on data and methods in a class (Not on **classes**) .



Accessibility Summary

Modifier on members in a class	Accessed from the **same class**	Accessed from the **same package**	Accessed from a **subclass**	Accessed from a **different package**
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–

Visibility Modifiers



A Subclass Cannot **Weaken** the Accessibility

- A subclass may **override** a `protected` method in its superclass and change its visibility to `public`.
- However, a subclass cannot **weaken** the accessibility of a method defined in the superclass.
  - ✓ For example, if a method is defined as `public` in the superclass, it must be defined as `public` in the subclass (cannot define it as `protected`, `default`, or `private`).

The `final` Modifier

- The **`final`** class cannot be extended:
  - ✓ e.g) The **`Math`** class and the **`String`** class

- The **`final`** variable is a constant:

```
final static double PI = 3.14159;
```

- The **`final`** method cannot be overridden by its subclasses.

```
Public final void m() {
    …
}
```

The `ArrayList` Class

You can create an array to store objects. But the array's **size** is **fixed** once the array is created. Java provides the `ArrayList` class that can be used to store an unlimited number of objects.

java.util.ArrayList<E>	
`+ArrayList()`	Creates an empty list.
`+add(o: E) : void`	Appends a new element o at the end of this list.
`+add(index: int, o: E) : void`	Adds a new element o at the specified index in this list.
`+clear(): void`	Removes all the elements from this list.
`+contains(o: Object): boolean`	Returns true if this list contains the element o.
`+get(index: int) : E`	Returns the element from this list at the specified index.
`+indexOf(o: Object) : int`	Returns the index of the **first matching** element in this list.
`+isEmpty(): boolean`	Returns true if this list contains no elements.
`+lastIndexOf(o: Object) : int`	Returns the index of the **last matching** element in this list.
`+remove(o: Object): boolean`	Removes the element o from this list.
`+size(): int`	Returns the number of elements in this list.
`+remove(index: int) : boolean`	Removes the element at the specified index.
`+set(index: int, o: E) : E`	Sets the element at the specified index.

Generic Type

- **`ArrayList`** is known as a generic class with a generic type E.
  - ✓ You can specify a concrete type to replace E when creating an ArrayList.

```
ArrayList<String> cities = new ArrayList<String>();
ArrayList<String> cities = new ArrayList<>();
```

Arrays and ArrayList

Operation	*Array*	*ArrayList*
Creating an array/ArrayList	`String[] a = new String[10]`	`ArrayList<String> list = new ArrayList<>();`
Accessing an element	`a[index]`	`list.get(index);`
Updating an element	`a[index] = "London";`	`list.set(index, "London");`
Returning size	`a.length`	`list.size();`
Adding a new element		`list.add("London");`
Inserting a new element		`list.add(index, "London");`
Removing an element		`list.remove(index);`
Removing an element		`list.remove(Object);`
Removing all elements		`list.clear();`

ArrayLists from/to Arrays

- Creating an ArrayList from an array of objects

```
String[] array = {"red", "green", "blue"};
ArrayList<String> list = new
                         ArrayList<>(Arrays.asList(array));
```

- Creating an array of objects from an ArrayList:

```
String[] array1 = new String[list.size()];
list.toArray(array1);
```

max and min in an ArrayList

```
String[] array = {"red", "green", "blue"};
System.out.pritnln(java.util.Collections.max(
    new ArrayList<String>(Arrays.asList(array)) );
```

```
String[] array = {"red", "green", "blue"};
System.out.pritnln(java.util.Collections.min(
  new ArrayList<String>(Arrays.asList(array)) );
```

Shuffling an ArrayList

```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};
ArrayList<Integer> list = new

ArrayList<>(Arrays.asList(array));
java.util.Collections.shuffle(list);
System.out.println(list);
```

The `MyStack` Classes

A stack to hold objects.

MyStack

MyStack
-list: ArrayList
+isEmpty(): boolean +getSize(): int +peek(): Object +pop(): Object +push(o: Object): void +search(o: Object): int

A list to store elements.

Returns true if this stack is empty.

Returns the number of elements in this stack.

Returns the top element in this stack.

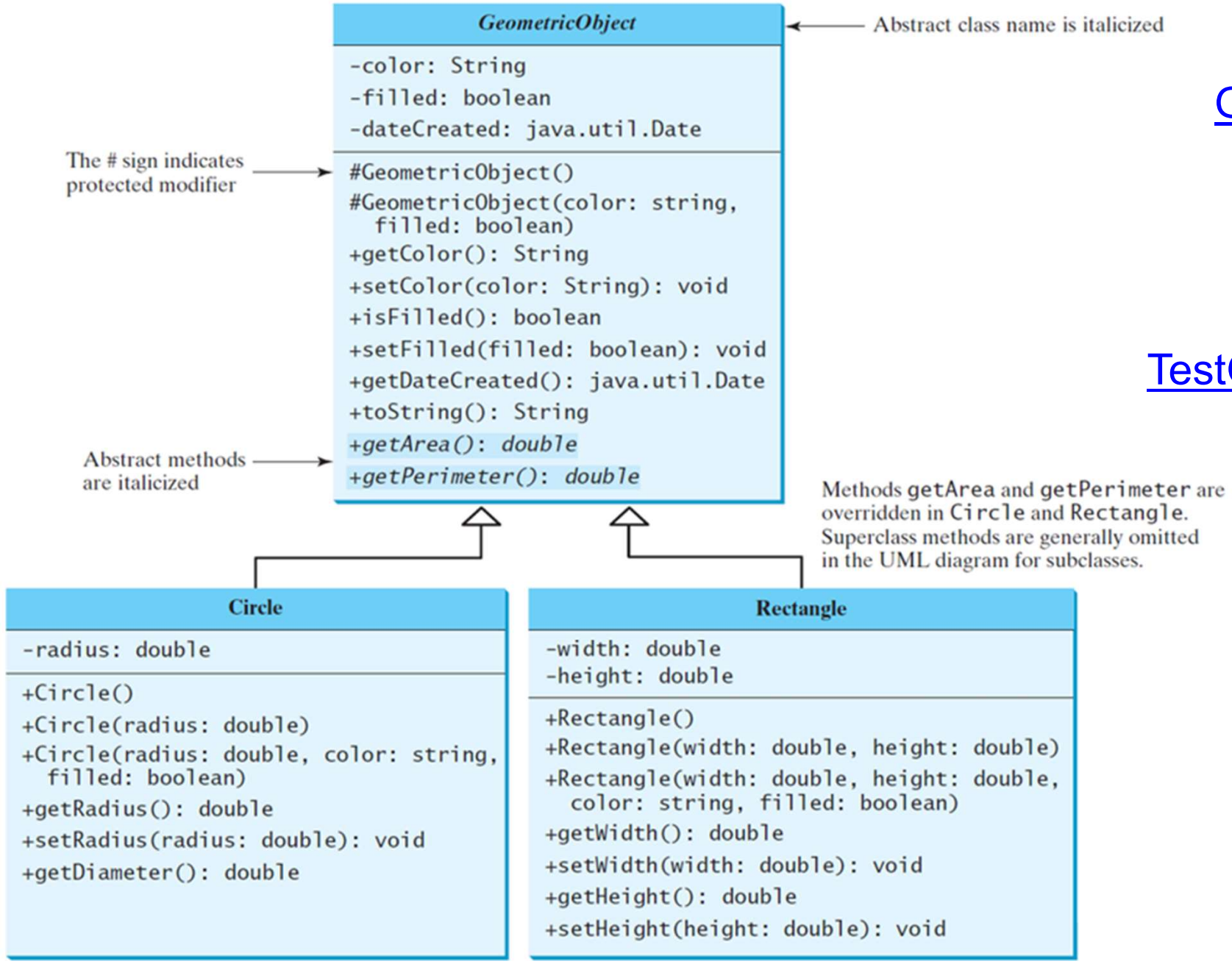
Returns and removes the top element in this stack.

Adds a new element to the top of this stack.

Returns the position of the first element in the stack from the top that matches the specified element.

ABSTRACT CLASS AND INTERFACE

Abstract Classes and Abstract Methods



GeometricObject

Circle

Rectangle

TestGeometricObject

Abstract Method in Abstract Class

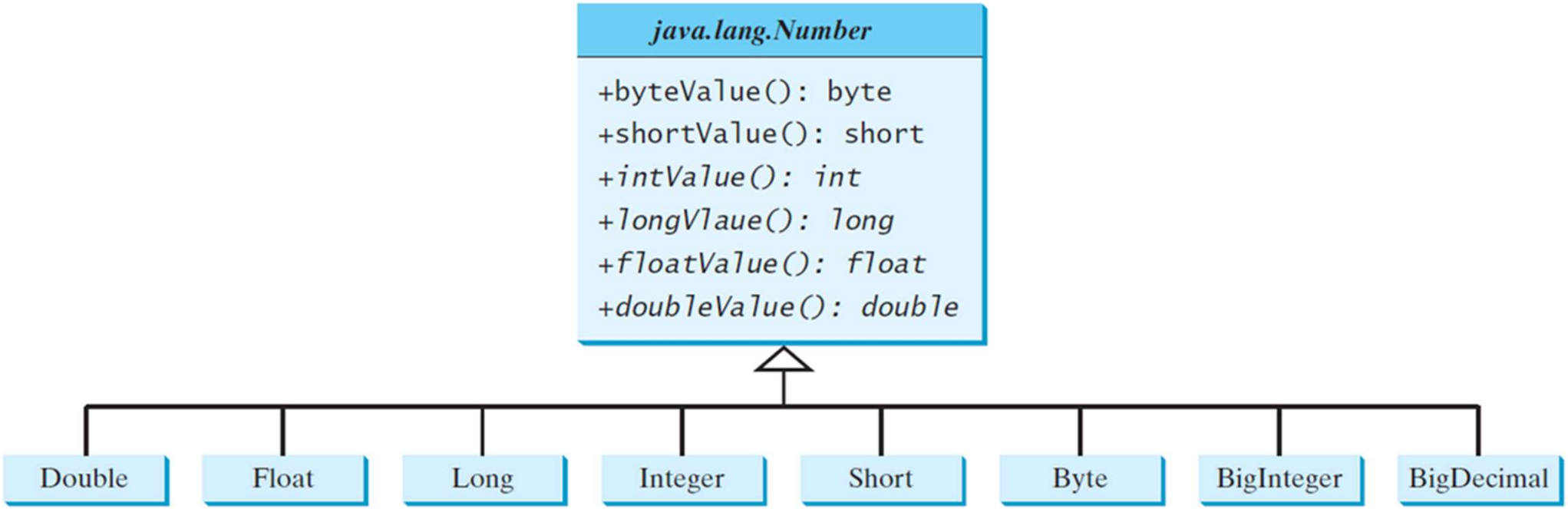
- Abstract method
  - ✓ cannot be contained in a nonabstract class
  - ✓ If a subclass of an abstract superclass does not implement all the abstract methods
    - The subclass must be defined abstract
    - In a nonabstract subclass extended from an abstract class,
      - All the abstract methods must be implemented, even if they are not used in the subclass

Abstract Class

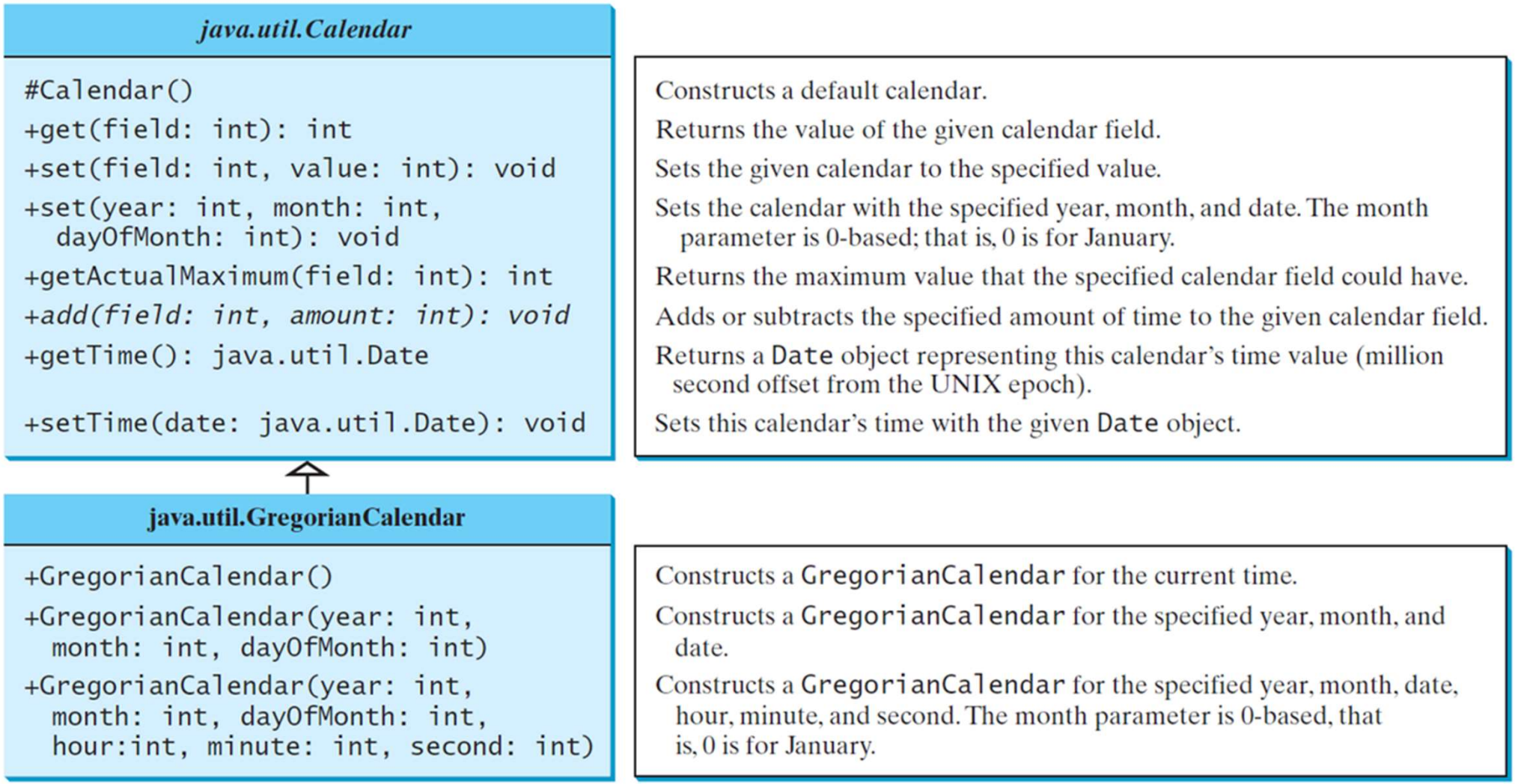
- Abstract class
  - ✓ cannot be instantiated using the new operator
    - BUT, you can still define its constructors
      - which are invoked in the constructors of its subclasses.
  - ✓ An abstract class can be used as a data type
    - Ex) `GeometricObject[] geo = new GeometricObject[10];`
  - ✓ A class that contains abstract methods must be abstract
  - ✓ It is possible to define an abstract class that contains no abstract methods
    - In this case, you cannot create instances of the class using the new operator. This class is used as a base class for defining a new subclass.
  - ✓ A subclass can be abstract even if its superclass is concrete
    - Ex) the Object class is concrete, but its subclasses, such as `GeometricObject`, may be abstract.

The Abstract Number Class

LargestNumbers



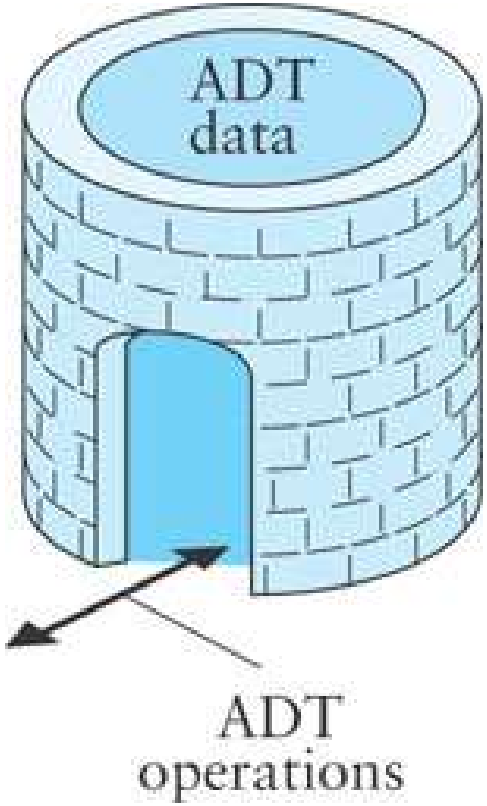
Abstract `Calendar` Class and Its `GregorianCalendar` Subclass



ABSTRACT DATE TYPE (ADT) AND INTERFACE

Abstract Date Type (ADT)

- An **encapsulation** of **data** and **methods**
  - ✓ The user need not know about the implementation of the ADT
- Allows for **reusable** code
  - ✓ A user interacts with the ADT **using** only **public** methods
- ADTs often are called ***data structures***
  - The Java `Collections` Framework provides implementations of common data structures



Interfaces

- specifies or describes an ADT to the applications programmer
  - ✓ the **methods** and the **actions** that they must perform
  - ✓ what **arguments**, if any, must be passed to each method
  - ✓ what **result** the method will return
- The interface can be viewed as a **contract** that guarantees how the ADT will function
  - ✓ The **interface** definition shows only **headings** for its methods ➔ **Abstract methods**
    - Each abstract method must be defined in a class that implements the interface
  - ✓ A class that implements the interface provides code for the ADT
    - **In addition** to implementing all data fields and methods in the interface,
      - ➢ data fields not in the implementation
      - ➢ methods not in the implementation
      - ➢ constructors (an interface cannot contain constructors because it cannot be instantiated)

Syntax:

```
public interface interfaceName {
    abstract method headings
    constant declarations
}
```

```
public interface Payable {
    public abstract double calcSalary();
    public abstract boolean salaried();
    public static final double DEDUCTIONS = 25.5;
}
```

- Constants are defined in the interface
- **DEDUCTIONS** are accessible in classes that implement the interface
- The keywords **public** and **abstract** are implicit in each abstract method definition
- **public static final** are implicit in each **constant** declaration

Example: ATM Interface

- An automated teller machine (ATM) enables a user to perform certain banking operations from a remote location. It must provide operations to
  - ✓ verify a user's Personal Identification Number (PIN)
  - ✓ allow the user to choose a particular account
  - ✓ withdraw a specified amount of money
  - ✓ display the result of an operation
  - ✓ display an account balance

- A **class** that **implements** an ATM must provide a method for each operation

```
public interface ATM {
    ... ...
}
```

the header indicates that an **interface** is being **declared**

```
public interface ATM {
   /** Verifies a user's PIN.
       @param pin The user's PIN   */
   boolean verifyPIN(String pin);

   /** Allows the user to select an
       account.
       @return a String representing
               the account selected   */
   String selectAccount();
  /** Withdraws a specified amount
       of money
       @param account The account
              from which the money
              comes
       @param amount The amount of
              money withdrawn
       @return whether or not the
              operation is
              successful
     */
   boolean withdraw(String account,
                    double amount);
```

```
   /** Displays the result of an
         operation
         @param account The account
                from which money was
                withdrawn
         @param amount The amount of
                money withdrawn
         @param success Whether or not
                the withdrawal took
                place
      */
     void display(String account,
                  double amount,
                  boolean success);

    /** Displays an account balance
         @param account The account
                selected
     */
     void showBalance(String account);
}
```

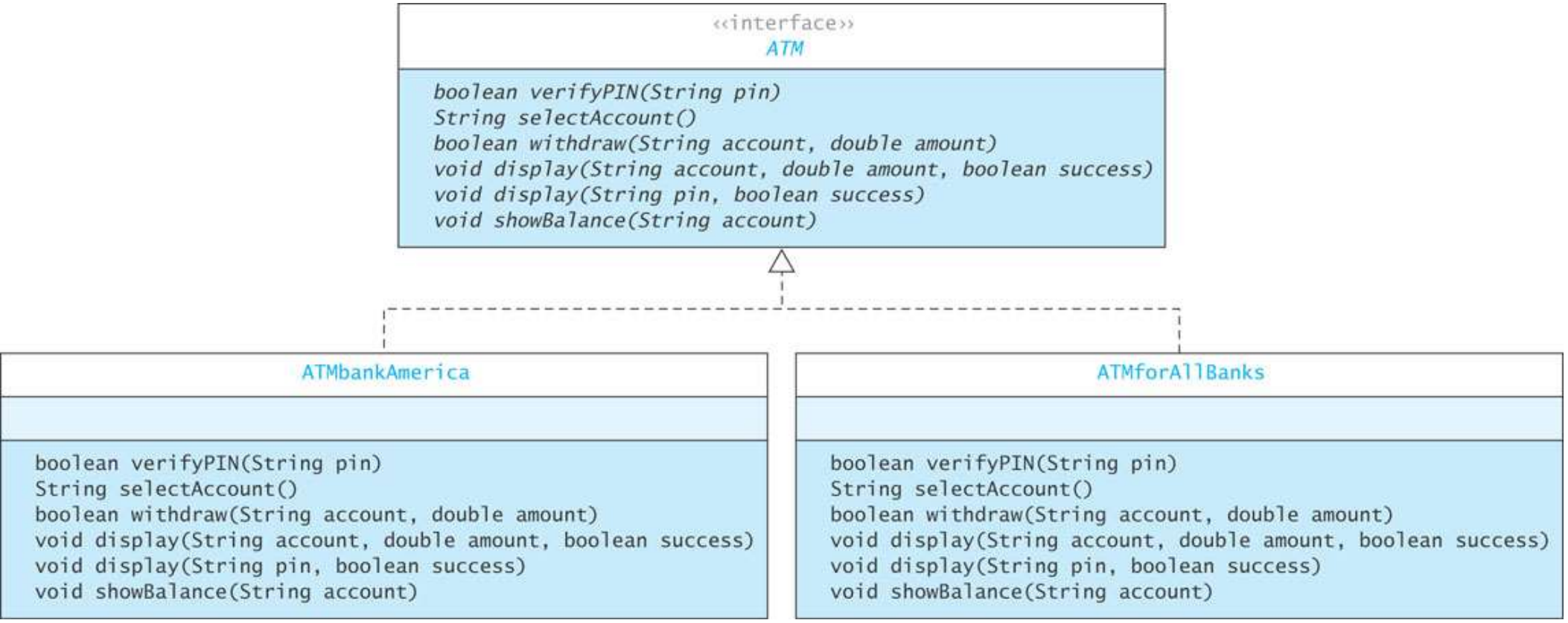
`implements` Clause

- For a class to implement an interface, it must end with the `implements` clause

```
public class ATMbankAmerica implements ATM
public class ATMforAllBanks implements ATM
```

- A class may implement more than one interface
  - ✓ Interface names are separated by commas

UML of Interface & Implementers



`implements` Clause: Pitfalls

- Java compiler verifies that a class defines all the abstract methods in its interface(s)
  - ✓ ➔ A **syntax error** will occur if a method is not defined or is not defined correctly:

```
Class ATMforAllBanks should be declared abstract; it does not
      define method verifyPIN(String) in interface ATM
```

- If a class contains an undefined abstract method
  - ✓ ➔ The compiler will require that the class be declared an abstract class
- You cannot instantiate an interface

```
ATM anATM = new ATM();     // invalid statement
```

Declaring a Variable of an Interface Type

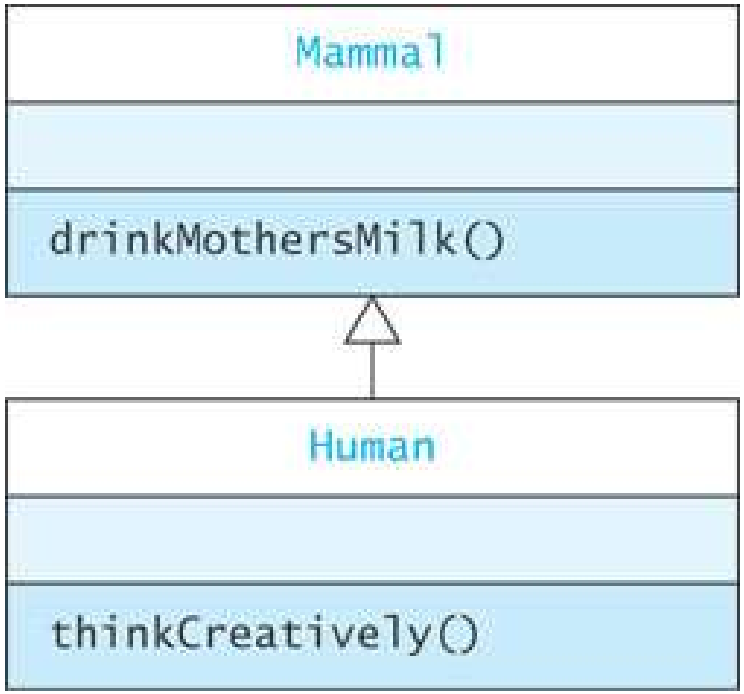
- While you cannot instantiate an interface, you can declare a **variable** that has an **interface type** ➔ **Polymorphism**

```
/* expected type */
ATMbankAmerica ATM0 = new ATMBankAmerica();
```

```
/* interface type */
ATM ATM1 = new ATMBankAmerica();
ATM ATM2 = new ATMforAllBanks();
```

Inheritance

- A **Human** *is a* **Mammal**
  - Mammal is the **superclass** of Human
    - Mammal has only method **`drinkMothersMilk()`**
    - Human has all the data fields and methods defined by Mammal
  - Human is a **subclass** of Mammal
    - Human may define other variables and methods that are not contained in Mammal
    - Human has method **`drinkMothersMilk`**`()` and **`thinkCreatively`**`()`



Inheriting from **Interfaces** VS **Classes**

- A class can ***extend* 0 or 1 superclass**
- An interface cannot extend a class
- A class or interface can ***implement* 0 or more interfaces**

Actual Class VS Abstract Class VS Interface

Property	Actual Class	Abstract Class	Interface
Instances (objects) of this can be created.	Yes	No	No
This can define instance variables and methods.	Yes	Yes	No
This can define constants.	Yes	Yes	Yes
The number of these a class can extend.	0 or 1	0 or 1	0
The number of these a class can implement.	0	0	Any number
This can extend another class.	Yes	Yes	No
This can declare abstract methods.	No	Yes	Yes
Variables of this type can be declared.	Yes	Yes	Yes