

Abstract Classes

Introduction to Programming and
Computational Problem Solving - 2

CSE 8B

Lecture 14

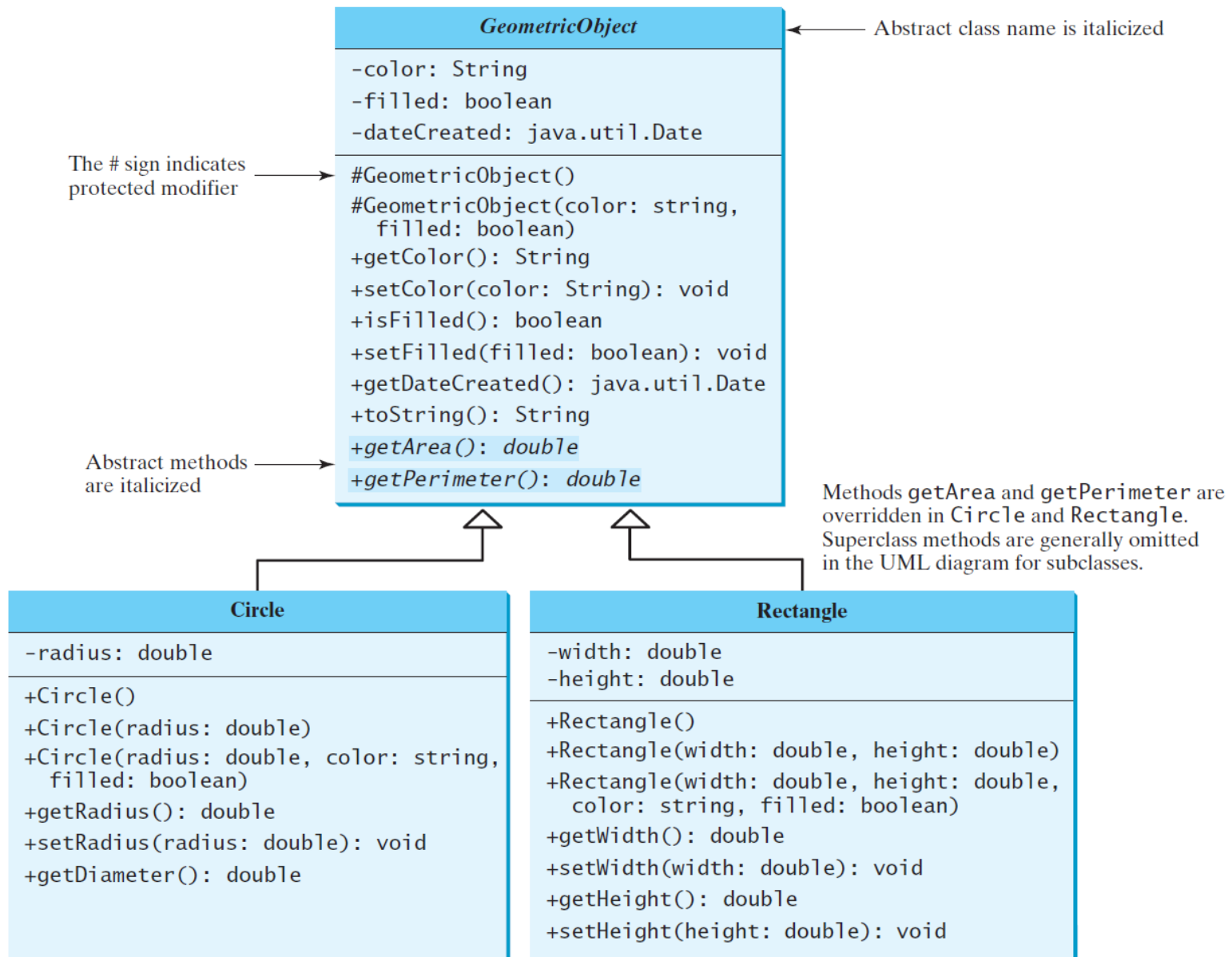
Announcements

- Assignment 7 is due Nov 17, 11:59 PM
- Quiz 7 is Nov 19
- Assignment 8 will be released Nov 17
 - Due Nov 24, 11:59 PM
- Educational research study
 - Nov 19, weekly survey
- Reading
 - Liang
 - Chapter 13

Abstract classes

- Remember, a superclass defines common behavior for **related** subclasses
 - Inheritance enables you to define a general class (i.e., a *superclass*) and later extend it to more specialized classes (i.e., *subclasses*)
- Sometimes, a superclass is so general it cannot be used to create objects
 - Such a class is called an *abstract class*
- An **abstract** class can contain abstract methods that are implemented in **concrete** subclasses
- Just like nonabstract classes, models **is-a** relationships
 - For example
 - Circle **is-a** GeometricObject
 - Rectangle **is-a** GeometricObject

Abstract class example



abstract modifier

- Abstract classes and abstract methods are denoted using the abstract modifier

- Example

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

    // Construct a default geometric object
    protected GeometricObject() {
        dateCreated = new java.util.Date();
    }

    // Construct a geometric object with color and filled value
    protected GeometricObject(String color, boolean filled) {
        dateCreated = new java.util.Date();
        this.color = color;
        this.filled = filled;
    }

    ...

    // Abstract method getArea
    public abstract double getArea();

    // Abstract method getPerimeter
    public abstract double getPerimeter();
}
```

Constructors in an abstract class are protected because they are only used by subclasses

Abstract methods are only allowed in abstract classes

- An 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 other words, 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**

An object cannot be created from an abstract class

- An abstract class cannot be instantiated using the `new` operator
- You can still define its constructors, which are invoked in the constructors of its subclasses
 - For example, the constructors of `GeometricObject` are invoked in the `Circle` class and the `Rectangle` class

An abstract class without any abstract methods

- Remember, a class containing any abstract methods must be abstract
- It is also possible to define an abstract class that does not contain any abstract methods
 - This class is used as a base class for defining a new subclass

Superclass of abstract class may be concrete

- A subclass can be abstract even if its superclass is concrete
 - For example, the `Object` class is concrete, but its subclasses (e.g., `GeometricObject`) may be abstract

Concrete method overridden to be abstract

- A subclass can override a method from its superclass to define it abstract
- **This is rare**, but useful when the implementation of the method in the superclass becomes invalid in the subclass
 - In this case, the subclass must be defined abstract

Abstract class as a data type

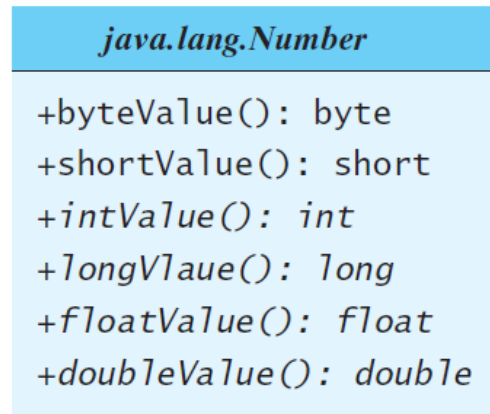
- Remember, an abstract class cannot be instantiated using the new operator
- However, an abstract class can be used as a data type

– Example

```
GeometricObject[] objects = new GeometricObject[2];  
objects[0] = new Circle();  
objects[1] = new Rectangle();
```

Abstract class example

- Number is an abstract superclass for the numeric wrapper classes (see lecture 9)



byteValue() and shortValue() call intValue() and cast result to byte and short, respectively

Double

Float

Long

Integer

Short

Byte

BigInteger

BigDecimal

Abstract class example

- `java.util.Calendar` is an abstract base class for extracting detailed information such as year, month, date, hour, minute, and second from a `Date` object
 - An instance of `java.util.Date` represents a specific instant in time with millisecond precision
- Subclasses of `Calendar` can implement specific calendar systems such as Gregorian calendar, Lunar Calendar, and Jewish calendar
- `GregorianCalendar` is a concrete subclass of the abstract class `Calendar`

Example: `GregorianCalendar` is a concrete subclass of the abstract class `Calendar`

java.util.Calendar

```
#Calendar()  
+get(field: int): int  
+set(field: int, value: int): void  
+set(year: int, month: int,  
    dayOfMonth: int): void  
+getActualMaximum(field: int): int  
+add(field: int, amount: int): void  
+getTime(): java.util.Date  
  
+setTime(date: java.util.Date): void
```

Constructs a default calendar.

Returns the value of the given calendar field.

Sets the given calendar to the specified value.

Sets the calendar with the specified year, month, and date. The month parameter is 0-based; that is, 0 is for January.

Returns the maximum value that the specified calendar field could have.

Adds or subtracts the specified amount of time to the given calendar field.

Returns a `Date` object representing this calendar's time value (million second offset from the UNIX epoch).

Sets this calendar's time with the given `Date` object.



java.util.GregorianCalendar

```
+GregorianCalendar()  
+GregorianCalendar(year: int,  
    month: int, dayOfMonth: int)  
+GregorianCalendar(year: int,  
    month: int, dayOfMonth: int,  
    hour: int, minute: int, second: int)
```

Constructs a `GregorianCalendar` for the current time.

Constructs a `GregorianCalendar` for the specified year, month, and date.

Constructs a `GregorianCalendar` for the specified year, month, date, hour, minute, and second. The month parameter is 0-based, that is, 0 is for January.

The `GregorianCalendar` Class

- Use `new GregorianCalendar()` to construct a default `GregorianCalendar` with the current time
- Use `new GregorianCalendar(year, month, date)` to construct a `GregorianCalendar` with the specified year, month, and date
 - The month parameter is 0-based (e.g., 0 is January)

Calendar is an abstract base class

- The `get(int field)` method defined in the `Calendar` class is useful to extract the date and time information from a `Calendar` object
- The fields are defined as constants

<i>Constant</i>	<i>Description</i>
<code>YEAR</code>	The year of the calendar.
<code>MONTH</code>	The month of the calendar, with 0 for January.
<code>DATE</code>	The day of the calendar.
<code>HOUR</code>	The hour of the calendar (12-hour notation).
<code>HOUR_OF_DAY</code>	The hour of the calendar (24-hour notation).
<code>MINUTE</code>	The minute of the calendar.
<code>SECOND</code>	The second of the calendar.
<code>DAY_OF_WEEK</code>	The day number within the week, with 1 for Sunday.
<code>DAY_OF_MONTH</code>	Same as <code>DATE</code> .
<code>DAY_OF_YEAR</code>	The day number in the year, with 1 for the first day of the year.
<code>WEEK_OF_MONTH</code>	The week number within the month, with 1 for the first week.
<code>WEEK_OF_YEAR</code>	The week number within the year, with 1 for the first week.
<code>AM_PM</code>	Indicator for AM or PM (0 for AM and 1 for PM).

The abstract add method

- The add method is abstract in the Calendar class because its implementation is dependent on a concrete calendar system
- `add(field, value)` adds the specific amount to a given field
 - Example
 - Add 7 days to the current time of the calendar
`add(Calendar.DAY_OF_MONTH, 7)`

Getting date/time information

```
public static void main(String[] args) {
    // Construct a Gregorian calendar for the current date and time
    Calendar calendar = new GregorianCalendar();
    System.out.println("Current time is " + new Date());
    System.out.println("YEAR: " + calendar.get(Calendar.YEAR));
    System.out.println("MONTH: " + calendar.get(Calendar.MONTH));
    System.out.println("DATE: " + calendar.get(Calendar.DATE));
    System.out.println("HOUR: " + calendar.get(Calendar.HOUR));
    System.out.println("HOUR_OF_DAY: " +
        calendar.get(Calendar.HOUR_OF_DAY));
    System.out.println("MINUTE: " + calendar.get(Calendar.MINUTE));
    System.out.println("SECOND: " + calendar.get(Calendar.SECOND));
    System.out.println("DAY_OF_WEEK: " +
        calendar.get(Calendar.DAY_OF_WEEK));
    System.out.println("DAY_OF_MONTH: " +
        calendar.get(Calendar.DAY_OF_MONTH));
    System.out.println("DAY_OF_YEAR: " +
        calendar.get(Calendar.DAY_OF_YEAR));
    System.out.println("WEEK_OF_MONTH: " +
        calendar.get(Calendar.WEEK_OF_MONTH));
    System.out.println("WEEK_OF_YEAR: " +
        calendar.get(Calendar.WEEK_OF_YEAR));
    System.out.println("AM_PM: " + calendar.get(Calendar.AM_PM));

    // Construct a calendar for December 11, 2021
    Calendar calendar1 = new GregorianCalendar(2021, 11, 11);
    String[] dayNameOfWeek = {"Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"};
    System.out.println("December 11, 2021, is a " +
        dayNameOfWeek[calendar1.get(Calendar.DAY_OF_WEEK) - 1]);
}
```

Class design guidelines

Coherence

- A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose
- A single entity with many responsibilities can be broken into several classes to separate responsibilities

Consistency

- Follow standard Java programming style and naming conventions
- Choose informative names for classes, data fields, and methods
 - Make names consistent
- Place the data declaration before the constructor, and place constructors before methods
- Provide a no-arg constructor (or document why the class does not support one)

Encapsulation

- A class should use the `private` modifier to hide its data from direct access by clients
- Provide getter methods and setter methods to provide users with access to the private data, but only to private data you want the user to see or to modify
- A class should also hide methods not intended for client use
- Make methods `protected` if they are intended for extenders of the class

Clarity and completeness

- A class should have a clear contract that is easy to explain and easy to understand
- Design a class that imposes no restrictions on how or when the user can use it
 - Design the properties in a way that lets the user set them in any order and with any combination of values
 - Design methods that function independently of their order of occurrence
- Methods should be defined intuitively without causing confusion
- You should not declare a data field that can be derived from other data fields
- A class should provide a variety of ways for customization through properties and methods that, together, are **minimal and complete**

Instance vs. static

- A variable or methods dependent on a specific instance of the class must be an instance variable or method
- A variable shared by all the instances of a class should be declared static
- A method not dependent on a specific instance should be defined as a static method
- Always reference static variables and methods from a class name to improve readability and avoid errors
- Do not initialize a static data field from a constructor parameter
 - Use a setter method to change the static data field

Inheritance vs. aggregation

- Use inheritance to model **is-a** relationships
- Use aggregation (and composition) to model **has-a** relationships

Next Lecture

- Interfaces
- Reading
 - Liang
 - Chapter 13