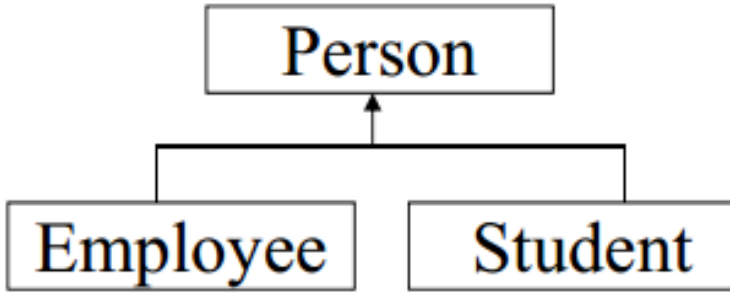
جامعة بيرزيت
BIRZEIT UNIVERSITY

Abstract Classes and Interfaces

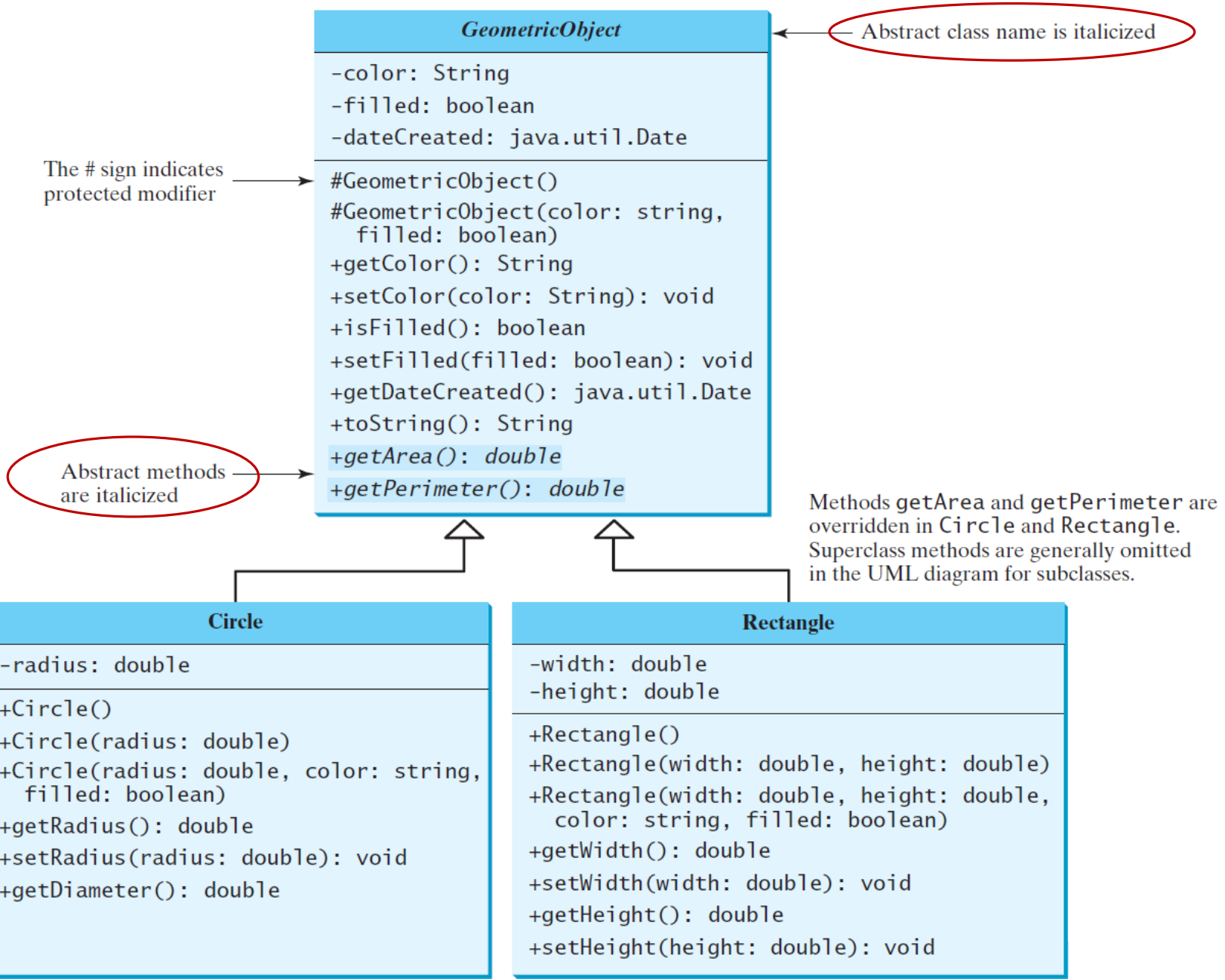
abstract Classes and Methods

- **Abstract classes**: some methods are **only** declared, but no **concrete** implementations are provided.
- Those methods called **abstract methods** and they need to be implemented by the extending classes.

```
abstract class Person {
   protected String name;
   . . .
   public abstract String getDescription() ;
   . . .
}
Class Student extends Person {
   private String major;
   . . .
   public String getDescription() {
         return name + " a student major in " + major;
   }
}
Class Employee extends Person {
   private float salary;
   . . .
   public String getDescription() {
         return name + " an employee with a salary of $ " + salary;
   }
}
```



abstract Classes and abstract Methods



abstract Method in abstract Class

❖ An **abstract** method **cannot** be contained in a non-abstract 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.

Object Can't be Created from abstract Class

❖ An **abstract** class **can't** be instantiated using the **new** operator, but you can still define its constructors, which are invoked in the constructors of its subclasses.

❖ For instance, the constructors of **GeometricObject** are invoked in the **Circle** class and the **Rectangle** class.

Abstract Class without Abstract Method

❖ A class that contains **abstract** methods **must** be **abstract**.

❖ However, 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.

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, such as **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 Type

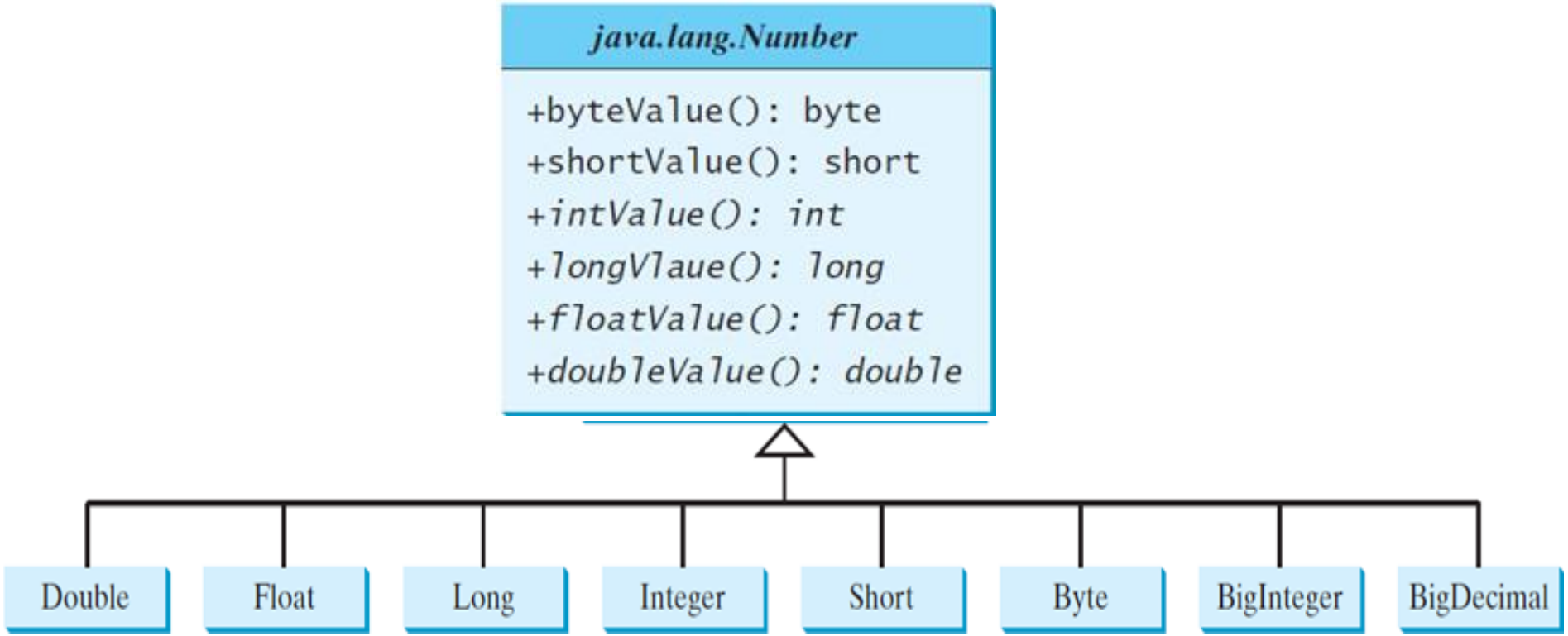
❖ You **can't** create an instance from an **abstract** class using the **new** operator, but an **abstract** class can be used as a data type.

❖ Therefore, the following statement, which creates an array whose elements are of **GeometricObject** type, is correct:

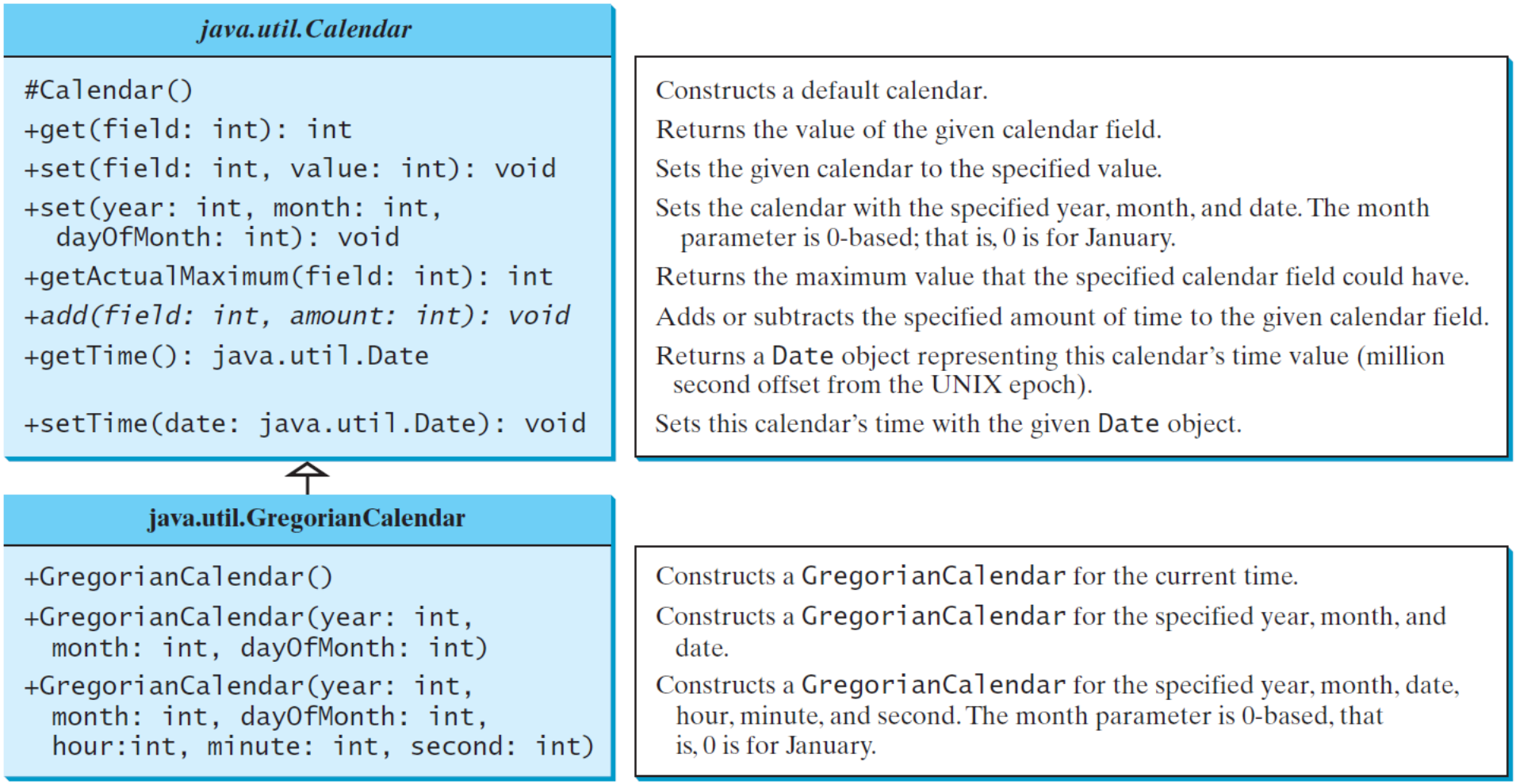
```
GeometricObject[]  geo = new GeometricObject[10];
```

Case Study:

The Abstract Number Class



The Abstract Calendar Class and Its GregorianCalendar subclass



GregorianCalendar subclass

- An instance of **java.util.Date** represents a specific instant in time with millisecond precision.
- **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.
- Subclasses of Calendar can implement specific calendar systems such as **Gregorian calendar**, **Lunar Calendar** and **Jewish calendar**.
- Currently, **java.util.GregorianCalendar** for the Gregorian calendar is supported in the Java API.

The GregorianCalendar Class

- You can 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**, i.e., 0 is for ***January***.

The get Method in Calendar Class

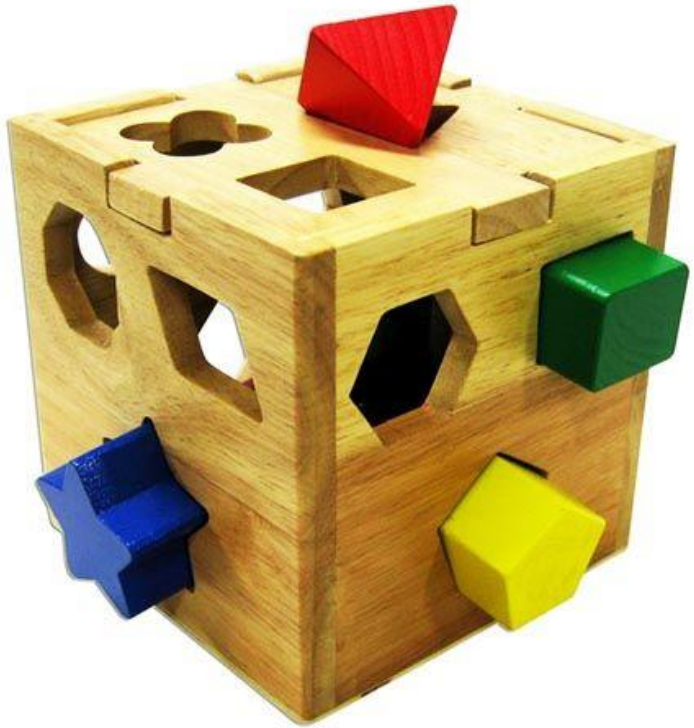
- The **get(int field)** method defined in the **Calendar** class is useful to extract the date and time information from a **Calendar** object. The fields are defined as constants, as shown in the following.

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

Interfaces

❖ An **interface** is a way to describe what classes should do, without specifying how they should do it.

❖ It is not a **class** but a set o **requirements** for classes that want to conform to the **interface**.



What is an interface?

❖ An **interface** is a **class-like** construct that contains **only constants** and **abstract** methods.

❖ In many ways, an **interface** is similar to an **abstract** class, but the intent of an interface is to specify **common behavior** for objects.

❖ For example, you can specify that the objects are ***comparable***, ***edible***, ***cloneable*** using appropriate interfaces.

Define an interface

❖ To distinguish an **interface** from a **class**, Java uses the following syntax to define an **interface**:

```
public interface InterfaceName {
    // constant declarations;
    // method signatures;
}
```

Example:

```
public interface Edible {
    /** Describe how to eat */
    public abstract String howToEat();
}
```

Interface is a Special Class

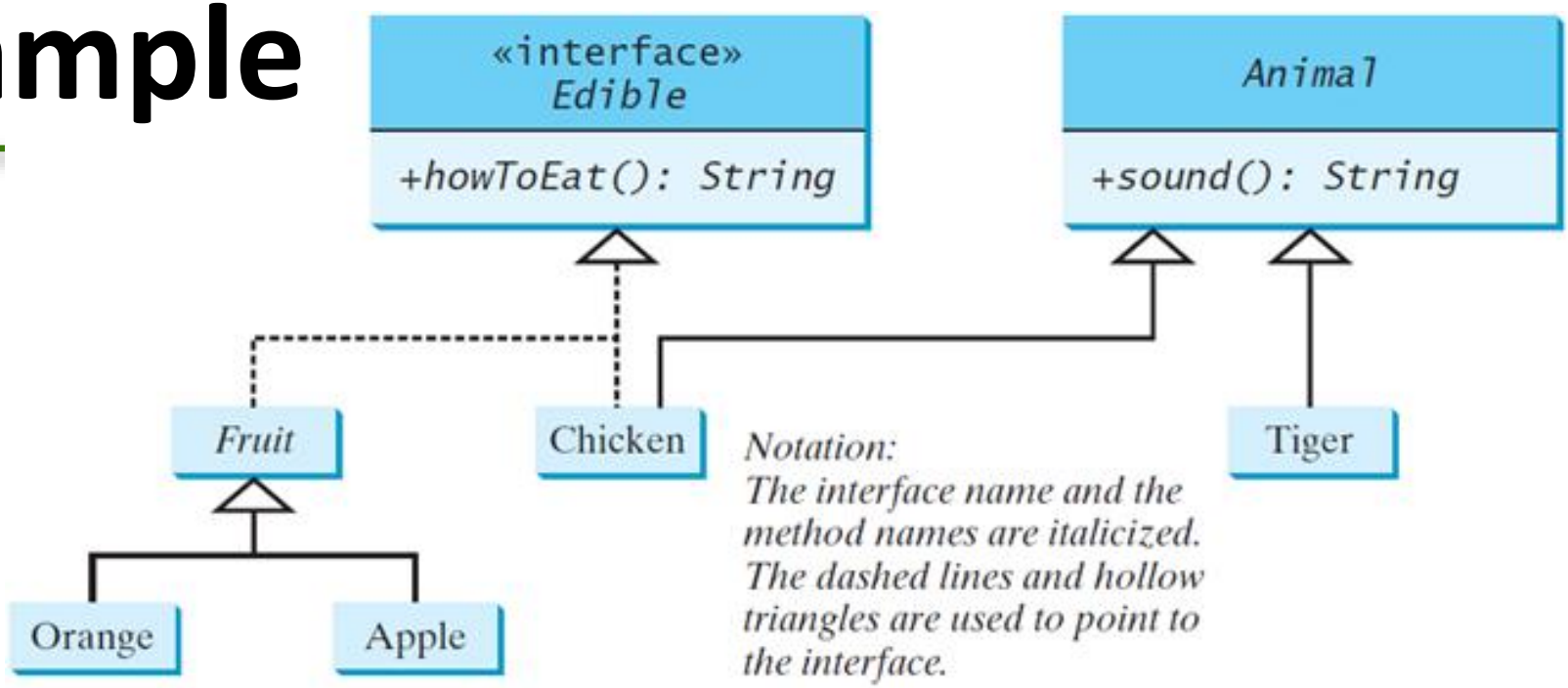
❖ An **interface** is treated like a special class in **Java**.

❖ Each **interface** is compiled into a separate **bytecode** file, just like a regular class.

❖ Like an **abstract** class, you **cannot** create an instance from an **interface** using the **new** operator, but in most cases you can use an **interface** more or less the same way you use an **abstract** class.

❖ For example, you can use an **interface** as a data type for variable, as the result of casting, and so on.

Example



❖ You can now use the **Edible** interface to specify whether an **object** is edible.

❖ This is accomplished by letting the class **implement** this interface using the **implements** keyword.

- For example, the classes **Chicken** and **Fruit** implement the **Edible** interface.

Omitting Modifiers in Interfaces

❖ All data fields are ***public final static*** and all methods are ***public abstract*** in an **interface**.

❖ For this reason, these modifiers can be **omitted**, as shown below:

```
public interface T1 {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

```
public interface T1 {
  int K = 1;

  void p();
}
```

❖ A constant defined in an **interface** can be accessed using syntax:

InterfaceName.CONSTANT_NAME

Example: The Comparable Interface

```
// This interface is defined in
// java.lang package
package java.lang;


public interface Comparable<E> {
    public int compareTo(E o);
}
```

Integer and BigInteger Classes

```
public class Integer extends Number
    implements Comparable<Integer> {
  // class body omitted

  @Override
  public int compareTo(Integer o) {
    // Implementation omitted
  }
}
```

```
public class BigInteger extends Number
    implements Comparable<BigInteger> {
  // class body omitted

  @Override
  public int compareTo(BigInteger o) {
    // Implementation omitted
  }
}
```

String and Date Classes

```
public class String extends Object
    implements Comparable<String> {
  // class body omitted

  @Override
  public int compareTo(String o) {
    // Implementation omitted
  }
}
```

```
public class Date extends Object
    implements Comparable<Date> {
  // class body omitted

  @Override
  public int compareTo(Date o) {
    // Implementation omitted
  }
}
```

Examples

```
Integer i1 = new Integer(3),    i2 = new Integer(3);

System.out.println(i1.compareTo( i2 ));
```

```
System.out.println("ABC".compareTo("ABE"));
```

```
Date date1 = new Date(2013, 1, 1);

Date date2 = new Date(2012, 1, 1);

System.out.println(date1.compareTo(date2));
```

instanceof

❖ Let **n** be an **Integer** object, **s** be a **String** object, and **d** be a **Date** object.

❖ All the following expressions are **true:**

```
n instanceof Integer
n instanceof Object
n instanceof Comparable
```

```
s instanceof String
s instanceof Object
s instanceof Comparable
```

```
d instanceof java.util.Date
d instanceof Object
d instanceof Comparable
```

The toString, equals, and hashCode Methods

- Each wrapper class overrides the **toString**, **equals**, and **hashCode** methods defined in the **Object** class.

- Since all the numeric wrapper classes and the Character class implement the **Comparable** interface, the **compareTo** method is implemented in these classes.

Generic sort Method

java.util.Arrays.sort(array)

- This method requires that the elements in an array are instances of **Comparable<E>**.

Extending Interfaces

- Interfaces support **multiple** inheritance: an **interface** can extend **more** than one **interface**.
- **Superinterfaces** and **subinterfaces**.
- Example:

```
public interface SerializableRunnable extends
    java.io.Serializable , Runnable {
            ...
}
```

Extending Interfaces – Constants

- If a **superinterface** and a **subinterface** contain two constants with the same name, then the one belonging to the superinterface is **hidden**:

```
interface X {
        int  val  = 1;
}
interface Y extends X {
        int  val  = 2;
        int sum = val  +  X.val;
}
```

Extending Interfaces – Methods

- If a declared method in a subinterface has the same signature as an inherited method **and** the same return type, then the new declaration **overrides** the inherited method in its superinterface.

- If the **only** difference is in the return type, then there will be a **compile-time error**.

The Cloneable Interface

❖ A class that implements the **Cloneable** interface is marked **cloneable**, and its objects can be cloned using the **clone()** method defined in the **Object** class.

❖ **clone** method returns a **new object** whose initial state is a **copy** of the current state of the object on which clone was invoked.

❖ Subsequent changes to the new clone object **should not** affect the state of the original object.

```
package java.lang;
public interface Cloneable {
}
```

Examples

❖ Many classes (e.g., **Date** and **Calendar**) implement **Cloneable**. Thus, the instances of these classes can be cloned. For example:

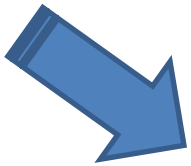
```
Calendar calendar = new GregorianCalendar(2003, 2, 1);
Calendar calendarCopy = (Calendar)calendar.clone();
System.out.println("calendar == calendarCopy is " +
                                    (calendar == calendarCopy));
System.out.println("calendar.equals(calendarCopy) is " +
                                    calendar.equals(calendarCopy));
```

```
calendar == calendarCopy is false
calendar.equals(calendarCopy) is true
```

Implementing Cloneable Interface

❖ To define a custom class that implements the **Cloneable** interface, the class **must** override the **clone()** method in the **Object** class.

❖ The following code defines a class named **House** that implements **Cloneable** and **Comparable**.



```
public  class House implements Cloneable, Comparable<House> {
  private  int  id;
  private  double  area;
  private  java.util.Date   whenBuilt;

  public House(int id, double  area) {
    this.id = id;
    this.area = area;
    whenBuilt = new java.util.Date();
  }

  public int getId()       {       return id;     }

  public double getArea()       {    return area;  }

  public java.util.Date   getWhenBuilt()   {    return whenBuilt;    }
```

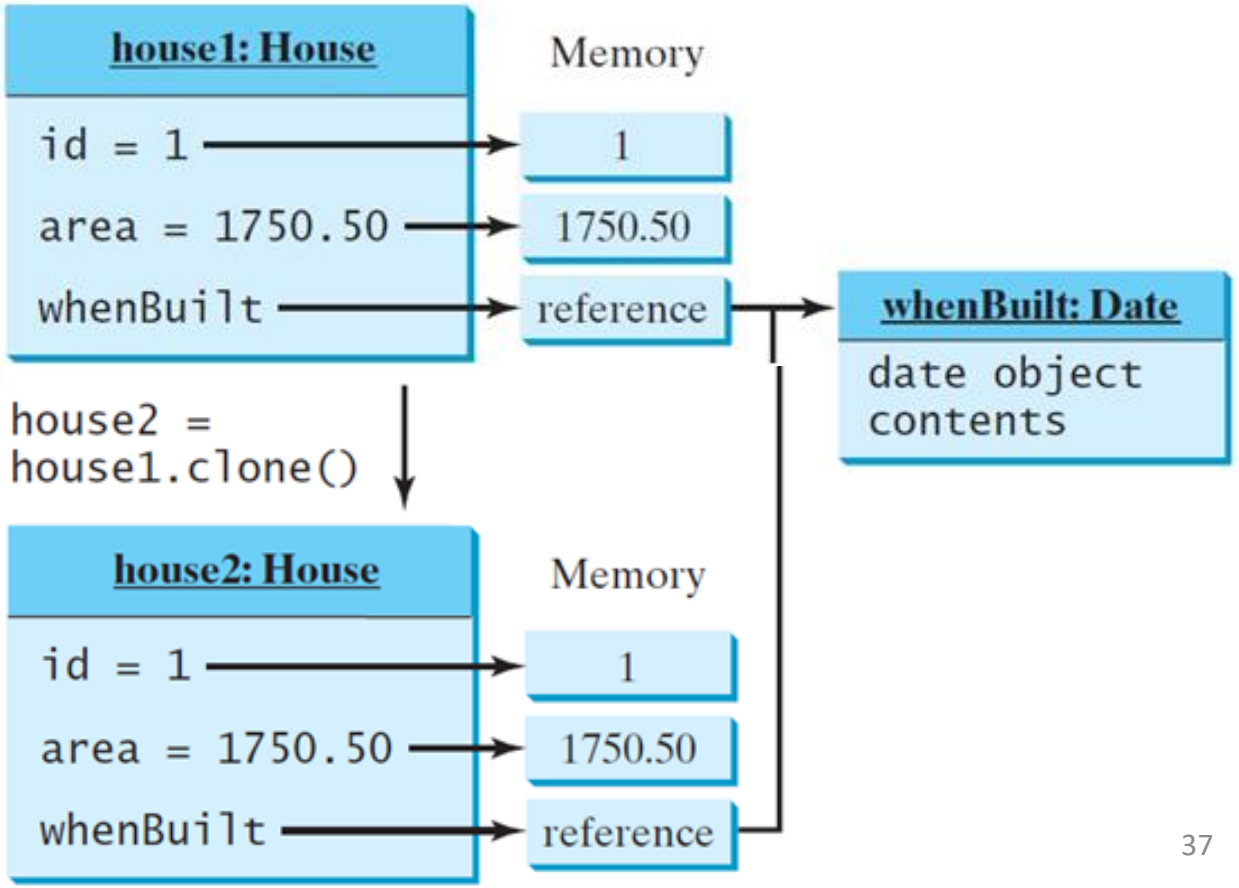
```
@Override // Override the clone method defined in the Object class
  public Object clone() {
        return  super.clone();
   }

  @Override // Implement the compareTo method defined in Comparable
  public int  compareTo(House  o) {
       if (area  >  o.area)
          return 1;
       else if (area < o.area)
          return -1;
       else
          return 0;
  }
}
```

Shallow vs. Deep Copy

House **house1** = new House(1, 1750.50);

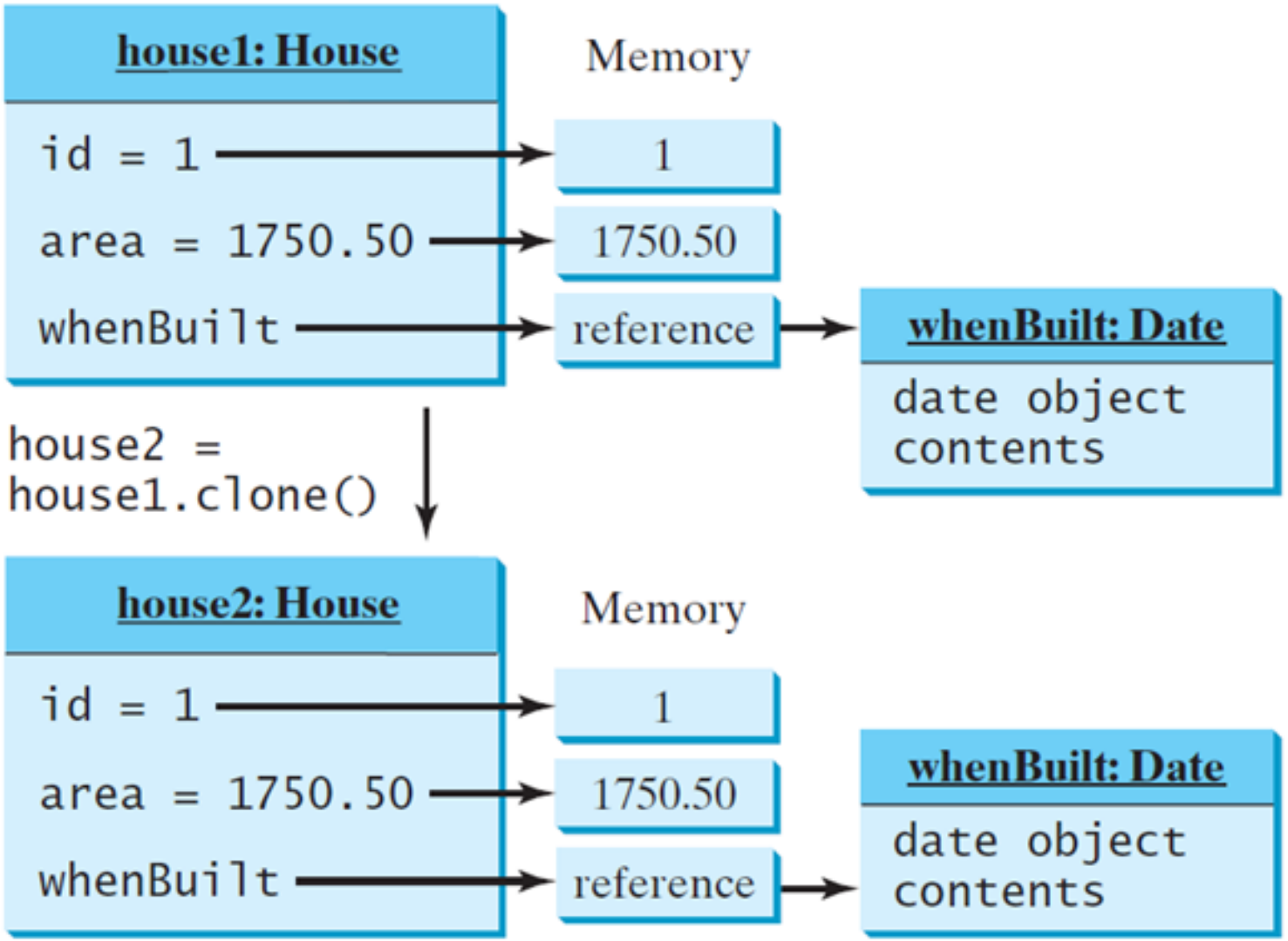
House **house2** = (House)house1.**clone**();

Shallow Copy



Shallow vs. Deep Copy

Deep Copy



Interfaces vs. Abstract Classes

❖ In an **interface**, the data must be **constants**; an **abstract** class can have all types of data.

❖ Each method in an **interface** has only a signature without implementation; an **abstract** class can have concrete methods.

	Variables	Constructors	Methods
Abstract class	No restrictions	Constructors are invoked by subclasses through constructor chaining. An abstract class **cannot** be instantiated using the new operator.	No restrictions.
Interface	All variables must be **public static final**	No constructors. An interface **cannot** be instantiated using the new operator.	All methods **must** be public abstract instance methods

Interfaces vs. Abstract Classes cont.

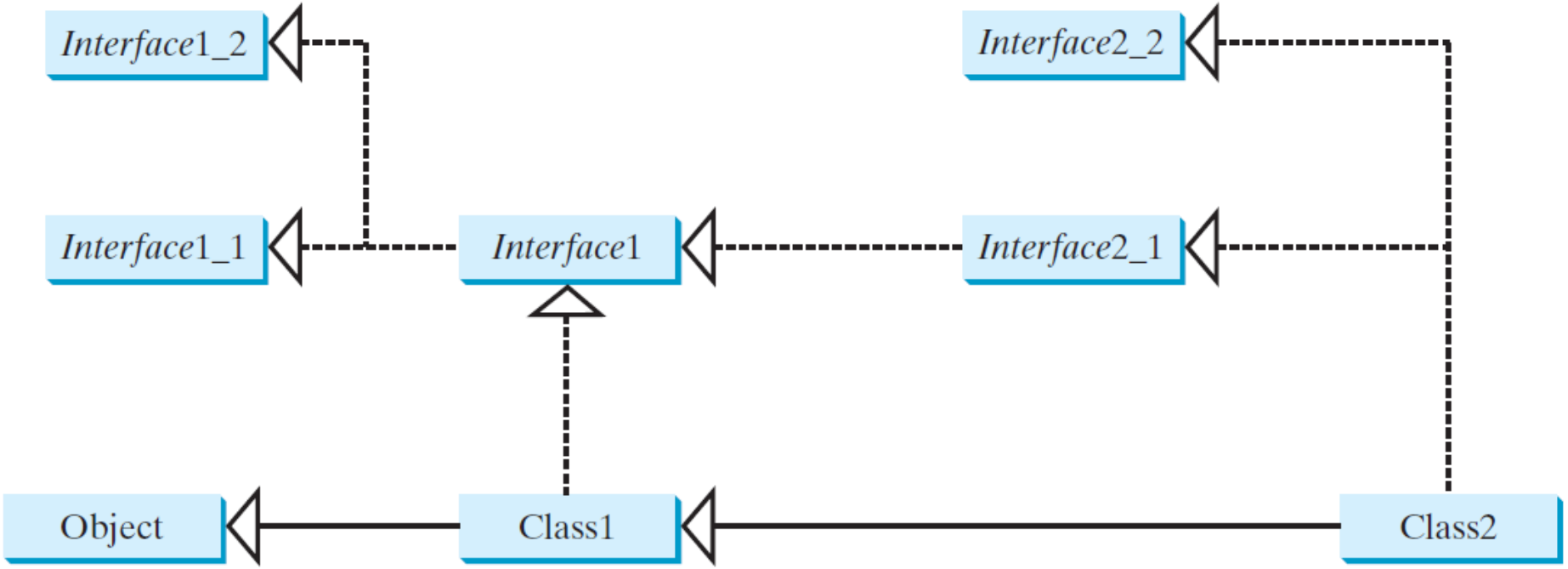
❖ All classes share a single root, the **Object** class, but there is no single root for interfaces.

❖ Like a **class**, an **interface** also defines a type. A variable of an **interface** type can reference any instance of the class that implements the **interface**.

❖ If a **class** extends an **interface**, this interface plays the same role as a superclass.

❖ You can use an **interface** as a data type and cast a variable of an **interface** type to its subclass, and vice versa.

instanceof



- Suppose that **c** is an instance of **Class2**.
- **c** is also an instance of **Object**, **Class1**, **Interface1**, **Interface1_1**, **Interface1_2**, **Interface2_1**, and **Interface2_2**.

Caution: conflict interfaces

- In rare occasions, a class may implement two interfaces with conflict information (e.g., two same constants with different values or two methods with same signature but different return type). This type of errors will be detected by the compiler.

Whether to use an interface or a class?

- Abstract classes and interfaces can both be used to model common features.
- How do you decide whether to use an interface or a class?
- In general, a strong is-a relationship that clearly describes a parent-child relationship should be modeled using classes.
- For example, a staff member is a person.

Whether to use an interface or a class?

- A weak is-a relationship, also known as an is-kind-of relationship, indicates that an object possesses a certain property.
- A weak is-a relationship can be modeled using interfaces.
- For example, all strings are comparable, so the String class implements the Comparable interface.
- You can also use interfaces to circumvent single inheritance restriction if multiple inheritance is desired.
- In the case of multiple inheritance, you have to design one as a superclass, and others as interface.