Abstract Classes and Interfaces Chapter

Contents

[Abstract classes 1](#_Toc35523021)

[Abstract methods 3](#_Toc35523022)

[Interfaces 3](#_Toc35523023)

[Comparable Interface 4](#_Toc35523024)

[compareTo method 5](#_Toc35523025)

[Cloneable Interface 6](#_Toc35523026)

[clone method 6](#_Toc35523027)

[Difference between Abstract Class and Interface in Java 8](#_Toc35523028)

[Design Guidelines 9](#_Toc35523029)

[Cohesion 9](#_Toc35523030)

[Consistency 9](#_Toc35523031)

[Encapsulation 9](#_Toc35523032)

[Clarity 9](#_Toc35523033)

[Completeness 9](#_Toc35523034)

[Instance vs Static 9](#_Toc35523035)

[Inheritance vs Aggregation 9](#_Toc35523036)

[Interfaces vs Abstract Class 9](#_Toc35523037)

Clarification:

* Superclasses define common state and behavior for *related* subclasses
* Subclasses become more specific and concrete with each new subclass
* Interfaces define common behavior for classes, related or not
* Abstract classes is a super class (related) that cannot create any specific instance
* Concrete classes = classes that are not abstract

**Remember two rules**:   
1) If the class is having few abstract methods and few concrete methods: declare it as **abstract class**.  
2) If the class is having only abstract methods: declare it as **interface**.

Abstract classes**:**

In the inheritance hierarchy, class become more specific and concrete with each new subclass.

The super class contains states and behaviors that are common to all subclasses.

Sometimes a superclass has common behavior to all subclass, but how to implement that behavior is different in each subclass. When this happens, that behavior/method becomes an abstract behavior/method.

When a class has an abstract behavior/method, then the class becomes abstract.

Please do not confuse this abstract class with OOP abstraction concept.

Utilizing the GeometricObject from the textbook, all geometric objects can calculate the area and calculate the perimeter, but each geometric object does so differently. This idea is what an abstract class is about, a behavior/method that is common to all but implement differently, hence GeometricObject becomes an abstract class.

Properties/ideas of abstract classes:

* You cannot create any specific instance of an abstract class, meaning you cannot instantiated using the new operator
  + **but** can be a data type, like this array of objects

Ex: GeometricObject [] objects = new GeometricObject [10];

objects[0] = new Circle();

Where GeometricObject is an abstract class and Circle is a concrete class and objects[0] is one element in the array of objects

* a subclass can be abstract even if the superclass is concrete
  + ***if*** the subclass does not implement (define/write the code) the abstract methods, then that subclass **has** to be an abstract class
* an abstract class can have concrete methods (methods with a body)
* since abstract classes allow the concrete methods, it may be stated as partial abstraction
  + still uses the abstract keyword
  + (interfaces provide 100% abstraction)
* Abstract classes have abstract methods without implementation
* Abstract classes do not have to have methods (other than the constructor)
* the constructor uses the protected visibility modifier; because it is used by subclasses only
  + (remember the inheritance chain of instantiation of objects) when an instance of the concrete subclass constructor is invoked the superclass's constructor is invoked to initialize data fields defined in the superclass
* Abstract classes **have** to state abstract: public abstract class NewObjectClass

Example:

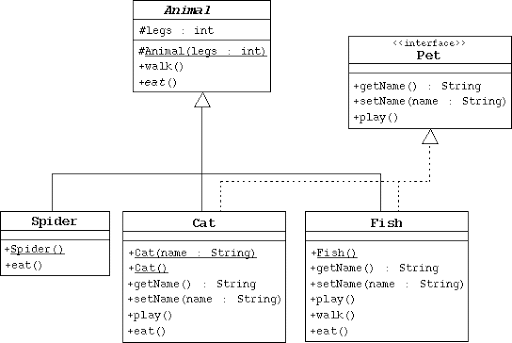
public abstract class Ball{

public abstract int hit(int batSpeed);

}

* If a class does not have abstract methods, it is not and abstract class but a concrete class
* in the UML diagram – abstract class names are *italicized* and abstract methods are *italicized*
  + **Note**: that the abstract methods do not have to be on the inherited, concrete class UML diagrams, it is understood that since those classes are concrete (the name is not italicized) those abstract methods are being defined in those classes

Example:



Abstract methods**:**

* are common methods used in the subclasses
* have no implementation/no body
  + the implementation is in the subclass that uses it = \*\*overridden\*\*
    - ***all*** abstract methods must be overridden
    - *if* the subclass does *not* implement the abstract method, that subclass *has* to be an abstract class
  + the JVM dynamically determines which method to invoke at runtime, depending on the actual object that invokes the method (dynamic polymorphism)
  + *Always end the declaration with a* ***semicolon****(;)*
* if there is an abstract method – then the class *has* to be abstract class
  + an abstract method *cannot* be contained in a non-abstract class
* if a subclass of an abstract class does not implement all the abstract methods, the subclass must be defined as abstract
* in the UML diagram – abstract methods are *italicized*
  + Superclass methods are generally omitted in the UML diagram for subclasses

Example:

public class BaseBall extends Ball{

public int hit(int batSpeed) {

// code that implements the hit method goes here }}

Interfaces**:**

Interfaces are used in OOP to define common behaviors amongst non-related and related classes.

Interface is a class-like structure that contains only constants and abstract methods.

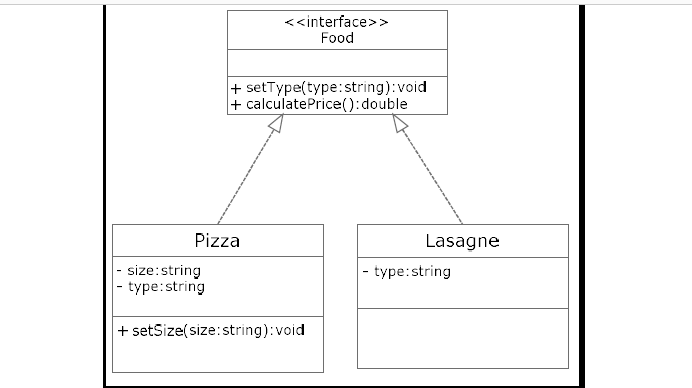
A Marker Interface is an interface with an empty body, it does not contain constants or methods

* It just specifies that a class has desirable properties

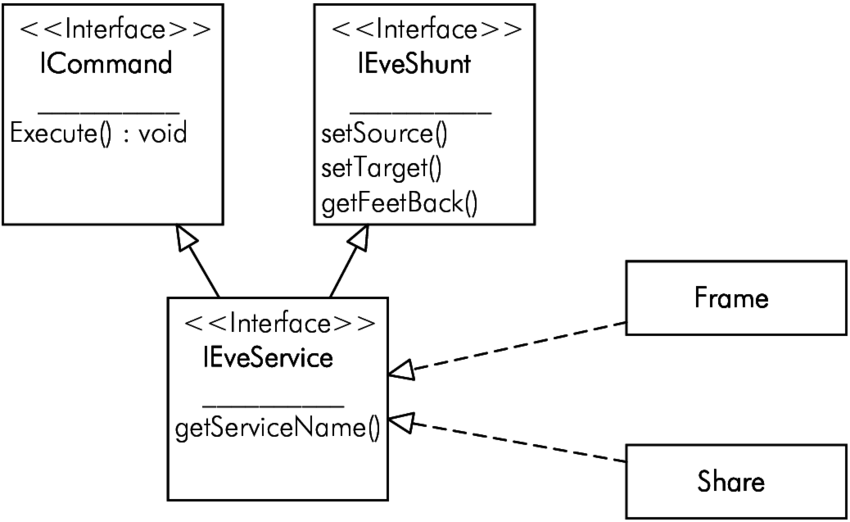
Points of Interfaces:

* The intent is to specify common behavior for objects of related classes or unrelated classes
* Each interface is compiles into a separate bytecode
* Uses the keyword interface, example: modifier interface InterfaceName
* You use interfaces the same way abstract classes are used
  + Meaning you can use an interface as a data type for a reference variable, cast, …
* You **cannot** create an instance from an interface using the new operator
* Modifiers can be omitted, since all methods are public abstract
* Provides another form of generic programming
* How Java works with/solves multiple inheritance
* UML notation: interface is encased in double chevrons (<<interface>>) and uses dashed lines and open triangles to denote the interface in the hierarchy

Example:



Example:



## Comparable Interface

The Comparable interface is an interface that allows objects to be compared to one another. For example Strings being sorted alphabetically.

It is defined as:

public interface Comparable<E>{

public int compareTo(E o);

}

The syntax of <E> and (E o) denotes the generics, meaning this is a generic interface. The idea behind generics is to be flexible, allowing any declared object to be used by that method.

<E> states generics (the industry standard is to use E or T)

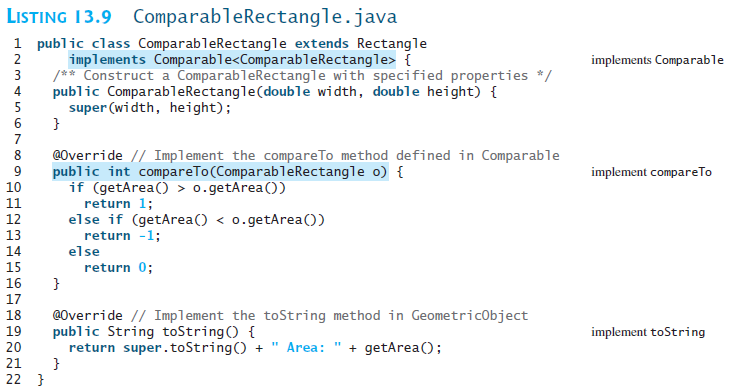
Then the datatype for that method is E and the variable identifier is o

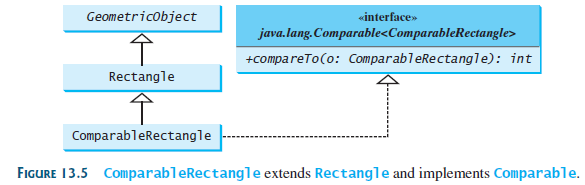
When this is invoked, the concrete datatype will replace the generic E.

### compareTo method

When you define the compareTo method for comparing your objects, you *must* utilize @override

Textbook examples:





## Cloneable Interface

The Cloneable interface specifies that an object can be cloned (an exact replica of the object is created).

Is defined:

public interface Cloneable{

}

* When a class implements the Cloneable interface, that class is marked as *cloneable*( meaning that objects of that class can be copied).

### clone method

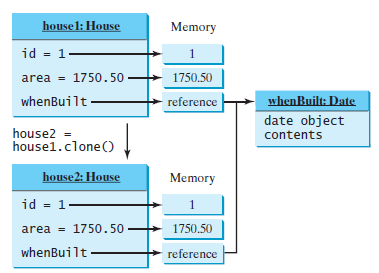
* The objects are copied by using the clone method in the Object class. The clone method creates a new object
  + How clone achieves this:

Copies each field/state from the original object to the target object

* + - When Java makes the copies, it just copies the content that is in main memory! So you will have to remember what is stored in main memory, what is stored there will result in one of two results, a shallow copy or a deep copy
    - If the data type is primitive, the values are copied as is
    - If the data type is object, the reference of the field is copied this results in a shallow copy
      * shallow copy: the references are copied
      * deep copy: the contents are copied
        + perform a deep copy

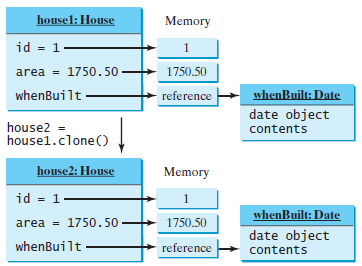
#### shallow copy

Memory representation of a **Shallow** copy, note that both objects reference the *same* whenBuilt Date; hence, these two objects are NOT a complete clone (due to the nature of how clone works):

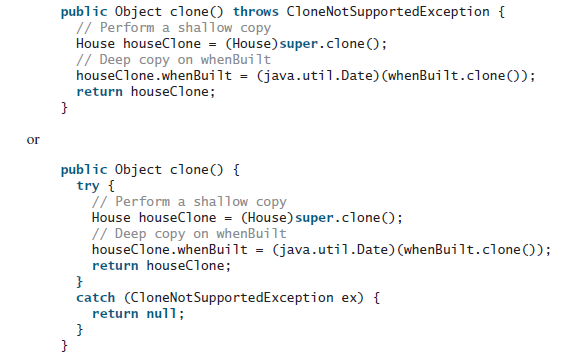


#### deep copy

Memory representation of a **Deep** copy, note that each object has their own whenBuilt Date; hence, these two objects are a complete clone.



The follows code example from the textbook shows two different ways to perform a deep copy.



* You must override the clone() method in the Object class
  + **Note** the throws CloneNotSupportedException has to used, either in the method header or in a try catch structure and you use @Override

@Override

public Object clone() throws CloneNotSupportedException {

# Difference between Abstract Class and Interface in Java

A class can implement multiple interfaces but only extend 1 superclass (inheritance)

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Variables** | **Constructors** | **Methods** |
| Abstract Class | No restrictions | Constructors are invoked by subclasses through constructor chaining. | No restrictions |
| Interface | No variables (from Java 8 and forward) | No constructors | Must be public abstract instance methods |

Strong *is-a* relationship = classes: abstract or concrete

Weak *is-a* or *is-kind-of* relationship = interfaces **(also called *can-do* relationship,** I prefer this idea, as the Interfaces are about behaviors, *can-do* reflects a behavior)

Both can be used to define common behavior.

Interfaces are more flexible than abstract classes because a subclass can only extend one superclass, but can implement numerous interfaces.

|  |  |  |
| --- | --- | --- |
|  | **Abstract Classes** | **Interfaces** |
| 1 | abstract class can extend only one class or one abstract class at a time | interface can extend any number of interfaces at a time |
| 2 | abstract  class  can extend from a class or from an abstract class | interface can extend only from an interface |
| 3 | abstract  class  can  have  both  abstract and concrete methods | interface can  have only abstract methods |
| 4 | A class can extend only one abstract class | A class can implement any number of interfaces |
| 5 | In abstract class keyword ‘abstract’ is mandatory to declare a method as an abstract | In an interface keyword ‘abstract’ is optional to declare a method as an abstract |
| 6 | abstract  class can have  protected , public and public abstract methods | Interface can have only public abstract methods i.e. by default |
| 7 | abstract class can have  static, final  or static final  variable with any access specifier | Interface  can  **only** have public static final (a constant) variable |

# Design Guidelines

|  |  |
| --- | --- |
| Cohesion | Coherent Purpose: a class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose  Separate Responsibility: a single entity with many responsibilities should be broken into several classes to separate the responsibilities |
| Consistency | Naming Conventions: follow the Java programming style and naming conventions  Style: popular style with object classes is to place the data declaration before the constructor and place the constructor before the methods  Naming Consistency: make the names consistent & follow Java's names for similar operations  No-arg Constructor: provide a no-arg constructor, if not, then document the reason it cannot do so; you can make a constructor private as in the Math class |
| Encapsulation | Encapsulate data fields: use the private modifier; provide getters for reading/accessing the data field; setters for updating data fields – only as needed |
| Clarity | Easy to Explain: a class should have a clear contract that is easy to explain and easy to understand; a single class should have cohesion, consistency, and encapsulation  Independent Methods: design a class that does not impose restrictions on how or when the user can use it - in any order and with any combination of values  Intuitive Meaning: should not cause confusion but should be intuitive to the user  Independent Properties: do not declare a data field that can be derived from other data fields |
| Completeness | The class should be useful in a wide range of applications and provide a variety of ways for customization through properties and methods |
| Instance vs Static | A variable or method that is dependent on a specific instance of the class must be an instance  A variable that is shared by all the instances of a class should be declared static  Use a mutator/setter method to change a static data field  A method that is not dependent on a specific instance should be defined as static  Always reference static variables and methods from a **class** name (not the reference name) to improve readability and avoid errors  A static variable or method *can be* invoked from an instance method, but an instance variable or method cannot be invoked from a static method  Meaning:  Instance **can** access/invoke static  Static **cannot** access/invoke instance |
| Inheritance vs Aggregation | Inheritance: *is-a* relationship  Aggregation: *has-a* relationship |
| Interfaces vs Abstract Class | Strong *is-a* relationship = class  Weak *is-a* relationship = interface  Also referred to as *"is-kind-of"* or *can-do* relationship |