**CHAPTER 13: ABSTRACT CLASSES AND INTERFACES**

* 1. **INTRODUCTION**

A superclass defines common behavior for related subclasses. An interface can be used to define common behavior for classes (including unrelated classes).

You can use the **java.util.Arrays.sort** method to sort an array of numbers or strings.

An interface is for defining common behavior for classes (including unrelated classes).

* 1. **ABSTRACT CLASSES**

An abstract class cannot be used to create objects. An abstract class can contain abstract methods that are implemented in concrete subclasses.

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

If you move from a subclass back up to a superclass, the classes become more general and less specific.

Class design should ensure a superclass contains common features of its subclasses.

Sometimes, a superclass is so abstract it cannot be used to create any specific instances. Such a class is referred to as an **abstract class**.

Abstract classes are like regular classes, but you cannot create instances of abstract classes using the **new** operator.

An abstract method is defined without implementation.

Its implementation is provided by the subclasses.

A class that contains abstract methods must be defined as abstract.

The constructor in the abstract class is defined as protected because it is used only by subclasses. When you create an instance of a concrete subclass, its superclass’s constructor is invoked to initialize data fields defined in the superclass.

The **GeometricObject** abstract class defines the common features (data and methods) for geometric objects and provides appropriate constructors.

* + 1. **Interesting Points about 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 as abstract.
* An 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.
* A class that contains abstract methods must be abstract. However, it is possible to define an abstract class that doesn’t contain any abstract methods. This abstract class is used as a base class for defining subclasses.
* A subclass can override a method from its superclass to define it as abstract. This is very unusual, but it is useful when the implementation of the method in the superclass becomes invalid in the subclass. In this case, the subclass must be defined as abstract.
* A subclass can be abstract even if its superclass is concrete.
* You cannot create an instance from an abstract class using the new operator, but an abstract class can be used as a data type.
  1. **The Abstract Number Class**

**Number** is an abstract superclass for numeric wrapper classes **BigInteger** and **BigDecimal**.

These classes have common methods **byteValue(), shortValue(), intValue(),** **longValue(),** **floatValue(),** and **doubleValue()** for returning a **byte**, **short**, **int**, l**ong**, **float**, and **double** value from an object of these classes. These common methods are actually defined in the Number class, which is a superclass for the numeric wrapper classes **BigInteger** and **BigDecimal.**

|  |  |
| --- | --- |
| java.lang.Number | |
| +byteValue(): byte | Returns this number as a byte. |
| +shortValue(): short | Returns this number as a short. |
| +intValue(): int | Returns this number as an int. |
| +longValue(): long | Returns this number as a long. |
| +floatValue(): float | Returns this number as a float. |
| +doubleValue(): double | Returns this number as a double. |

The Number class is an abstract superclass for **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **BigInteger**, and **BigDecimal**.

* 1. **Calendar and GregorianCalendar**

**GregorianCalendar** is a concrete subclass of the abstract class **Calendar**.

**java.util.Calendar** is an abstract base class for extracting detailed calendar information, such as the year, month, date, hour, minute, and second. Subclasses of Calendar can implement specific calendar systems, such as the Gregorian calendar, the lunar calendar, and the Jewish calendar. Currently, **java.util.GregorianCalendar** for the Gregorian calendar is supported in Java.

The **add** method is abstract in the Calendar class because its implementation is dependent on a concrete **calendar** system.

|  |  |
| --- | --- |
| java.util.Calendar | |
| #Calendar() | Constructs a default calendar. |
| +get(f ield: int): int | Returns the value of the given calendar field. |
| +set(f ield: int, value: int): void | Sets the given calendar to the specified value. |
| +set(year: int, month: int, dayOfMonth: int): void | Sets the calendar with the specified year, month, and date. The month parameter is 0-based; that is, 0 is for January. |
| +getActualMaximum(field: int): int | Returns the maximum value that the specified calendar field could have. |
| +add(f ield: int, amount: int): void | Adds or subtracts the specified amount of time to the given calendar field. |
| +getTime(): java.util.Date | Returns a Date object representing this calendar’s time value (million second offset from the UNIX epoch). |
| +setTime(date: java.util.Date): void | Sets this calendar’s time with the given Date object. |

|  |  |
| --- | --- |
| java.util.GregorianCalendar | |
| +GregorianCalendar() | Constructs a GregorianCalendar for the current time |
| +GregorianCalendar(year: int,  month: int, dayOfMonth: int) | Constructs a GregorianCalendar for the specified year, month, and date. |
| +GregorianCalendar(year: int,  month: int, dayOfMonth: int,  hour:int, minute: int, second: int) | 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 abstract **Calendar** class defines common features of various calendars.

**Field Constants in the Calendar Class:**

|  |  |
| --- | --- |
| 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). |

* 1. **INTERFACES**

An interface is a class-like construct for defining common operations for objects.

In many ways an interface is similar to an abstract class, but its intent is to specify common behavior for objects of related classes or unrelated classes.

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

**modifier interface InterfaceName {**

**/\*\* Constant declarations \*/**

**/\*\* Abstract method signatures \*/**

**}**

For example:

**public interface Edible {**

**/\*\* Describe how to eat \*/**

**public abstract String howToEat();**

**}**

An interface is treated like a special class in Java.

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

You can use an interface more or less the same way you use an abstract class.

As with an abstract class, you cannot create an instance from an interface using the new operator.

You can use the Edible interface to specify whether an object is edible. This is accomplished by letting the class for the object implement this interface using the implements keyword.

* 1. **The Comparable Interface**

The **Comparable** interface defines the **compareTo** method for comparing objects.

Java provides the Comparable interface to design a generic method to find the larger of two objects of the same type, such as two students, two dates, two circles, two rectangles, or two squares.

The interface is defined as follows:

**// Interface for comparing objects, defined in java.lang**

**package java.lang;**

**public interface Comparable {**

**public int compareTo(E o);**

**}**

The **compareTo** method determines the order of this object with the specified object **o** and returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than **o**.

The **Comparable** interface is a generic interface. The generic type **E** is replaced by a concrete type when implementing this interface.

* 1. **The Cloneable Interface**

The **Cloneable** interface specifies that an object can be cloned.

Often, it is desirable to create a copy of an object. To do this, you need to use the **clone** method and understand the **Cloneable** interface.

An interface contains constants and abstract methods, but the Cloneable interface is a special case.

The **Cloneable** interface in the **java.lang** package is defined as follows:

**package java.lang;**

**public interface Cloneable { }**

This interface is empty. An interface with an empty body is referred to as a **marker interface**. A marker interface is used to denote that a class possesses certain desirable properties. 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.

* Why is the **clone** method in the **Object** class defined protected, not public? Not every object can be cloned. The designer of Java purposely forces the subclasses to override it if an object of the subclass is cloneable.
* Why is the **clone** method not defined in the **Cloneable** interface? Java provides a native method that performs a shallow copy to clone an object.
  1. **Interfaces vs. Abstract Classes**

A class can implement multiple interfaces, but it can only extend one superclass.

An interface can be used more or less the same way as an abstract class, but defining an interface is different from defining an abstract class.

The following table shows the difference:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Variables | Constructors | Methods |
| Abtract 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. | May contain public abstract instance methods, public default, and public static methods. |

Java allows only single inheritance for class extension, but allows multiple extensions for interfaces.

An interface can inherit other interfaces using the **extends** keyword. Such an interface is called a **subinterface**.

For example, **NewInterface** in the following code is a **subinterface** of **Interface1**, . . . , and **InterfaceN**.

**public interface NewInterface extends Interface1, ... , InterfaceN {**

**// constants and abstract methods**

**}**

A class implementing **NewInterface** must implement the abstract methods defined in **NewInterface**, **Interface1**, . . . , and **InterfaceN**.

An interface can extend other inter faces, but not classes.

A class can extend its superclass and implement multiple interfaces.

**Note**: Class names are nouns. Interface names may be adjectives or nouns.

* 1. **The Rational Class**

A rational number has a numerator and a denominator in the form a/b, where a is the numerator and b the denominator.

A rational number cannot have a denominator of 0, but a numerator of 0 is fine.

Java provides data types for integers and floating-point numbers, but not for rational numbers.

Since rational numbers share many common features with integers and floating-point numbers, and **Number** is the root class for numeric wrapper classes, it is appropriate to define **Rational** as a subclass of **Number**.

Since rational numbers are comparable, the **Rational** class should also implement the **Comparable** interface.

|  |  |
| --- | --- |
| Rational | |
| –numerator: long  –denominator: long | The numerator of this rational number.  The denominator of this rational number. |
| +Rational() | Creates a rational number with numerator 0 and denominator 1. |
| +Rational(numerator: long, denominator: long) | Creates a rational number with a specified numerator and denominator. |
| +getNumerator(): long | Returns the numerator of this rational number. |
| +getDenominator(): long | Returns the denominator of this rational number. |
| +add(secondRational: Rational): Rational | Returns the addition of this rational number with another. |
| +subtract(secondRational: Rational): Rational | Returns the subtraction of this rational number with another. |
| +multiply(secondRational: Rational): Rational | Returns the multiplication of this rational number with another. |
| +divide(secondRational: Rational): Rational | Returns the division of this rational number with another. |
| +toString(): String | Returns a string in the form “numerator/denominator.” Returns the numerator if denominator is 1. |
| –gcd(n: long, d: long): long | Returns the greatest common divisor of n and d. |

The properties, constructors, and methods of the **Rational** class are illustrated in UML.

**Note:** The getter methods for the properties **numerator** and **denominator** are provided in the **Rational** class, but the setter methods are not provided, so, once a Rational object is created, its contents cannot be changed.

The **Rational** class is immutable.

The **String** class and the wrapper classes for primitive-type values are also immutable.

**THE END!**