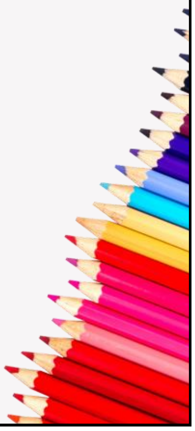
Chapter 13
Abstract Classes and Interfaces

Motivations

- You can use the java.util.Arrays.sort method to sort an array of numbers or strings. Can you apply the same sort method to sort an array of geometric objects?
- In order to write such code, you have to know about interfaces.
- An interface is for defining common behavior for classes (including unrelated classes). Before discussing interfaces, we introduce a closely related subject: abstract classes.



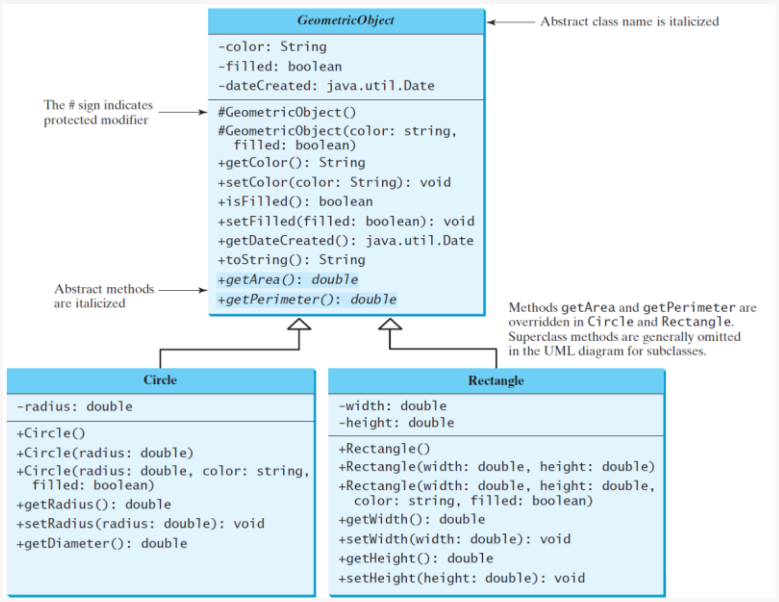
Objectives

- To design and use abstract classes (§13.2).
- To generalize numeric wrapper classes, BigInteger, and BigDecimal using the abstract Number class (§13.3).
- To process a calendar using the Calendar and GregorianCalendar classes (§13.4).
- To specify common behavior for objects using interfaces (§13.5).
- To define interfaces and define classes that implement interfaces (§13.5).
- To define a natural order using the Comparable interface (§13.6).
- To make objects cloneable using the Cloneable interface (§13.7).
- To explore the similarities and differences among concrete classes, abstract classes, and interfaces (§13.8).
- To design the Rational class for processing rational numbers (§13.9).
- To design classes that follow the class-design guidelines (§13.10).

13.2 Abstract Classes

- 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 that a superclass contains common features of its subclasses. Sometimes a superclass is so abstract that it cannot be used to create any specific instances. Such a class is referred to as an *abstract class.*
- *An abstract class cannot be used to create objects. An abstract class can contain abstract methods, which are implemented in concrete subclasses.*

Abstract Classes and Abstract Methods


GeometricObject
Abstract class name is italicized
-color: String
-filled: boolean
-dateCreated: java.util.Date
The # sign indicates protected modifier
#GeometricObject()
#GeometricObject(color: string, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
+getArea(): double
+getPerimeter(): double
Abstract methods are italicized
Methods getArea and getPerimeter are overridden in Circle and Rectangle. Superclass methods are generally omitted in the UML diagram for subclasses.
Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: string, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getDiameter(): double
Rectangle
-width: double
-height: double
+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double, color: string, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void

GeometricObject

Circle

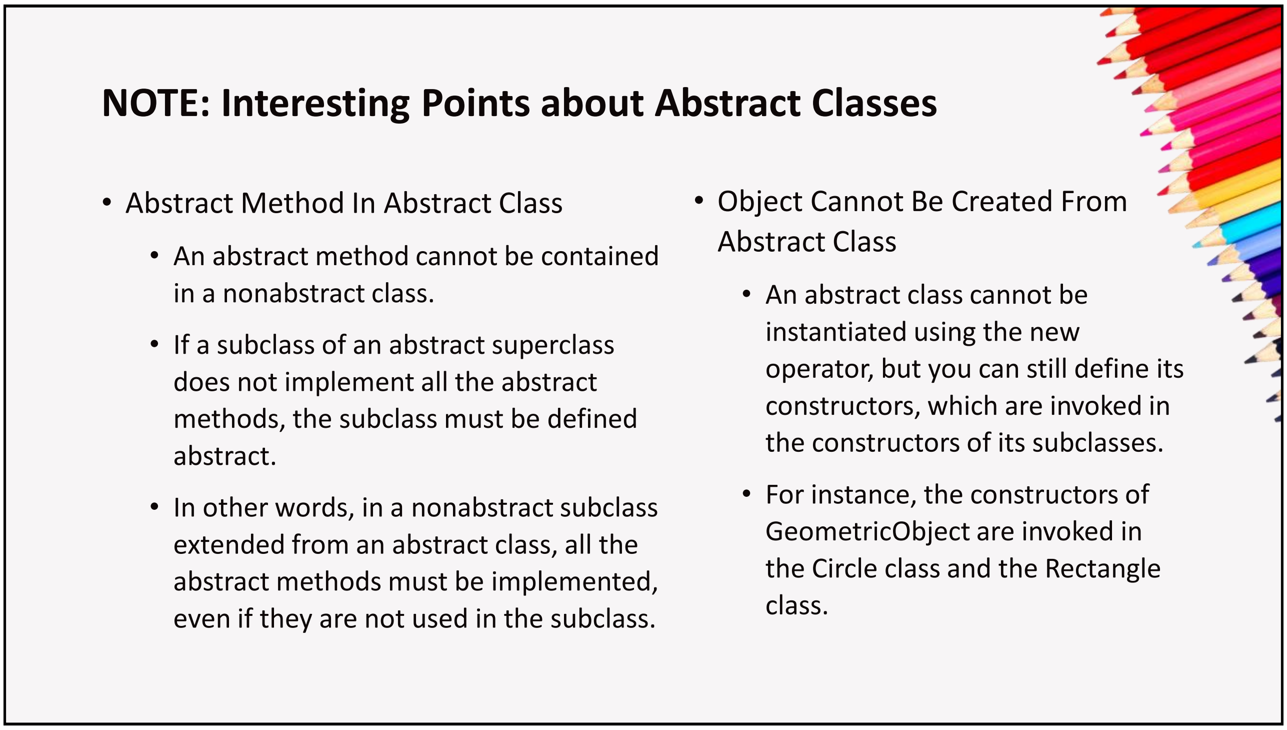
Rectangle

TestGeometricObject

Run

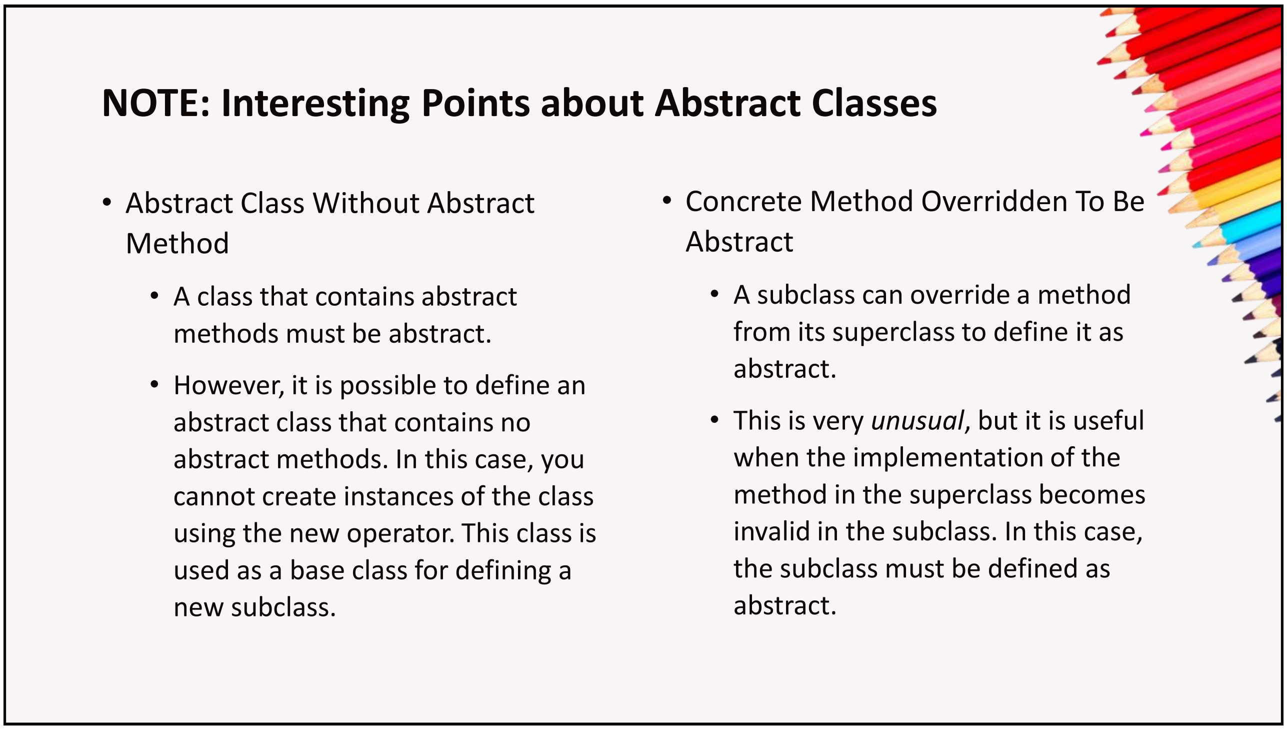
Abstract Classes and Abstract Methods

- 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.



NOTE: Interesting Points about Abstract Classes

- Abstract Method In Abstract Class
  - 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.
- Object Cannot Be Created From Abstract Class
  - 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.
  - For instance, the constructors of GeometricObject are invoked in the Circle class and the Rectangle class.



NOTE: Interesting Points about Abstract Classes

- 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.
- Concrete Method Overridden To Be Abstract
  - 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.

NOTE: Interesting Points about Abstract Classes

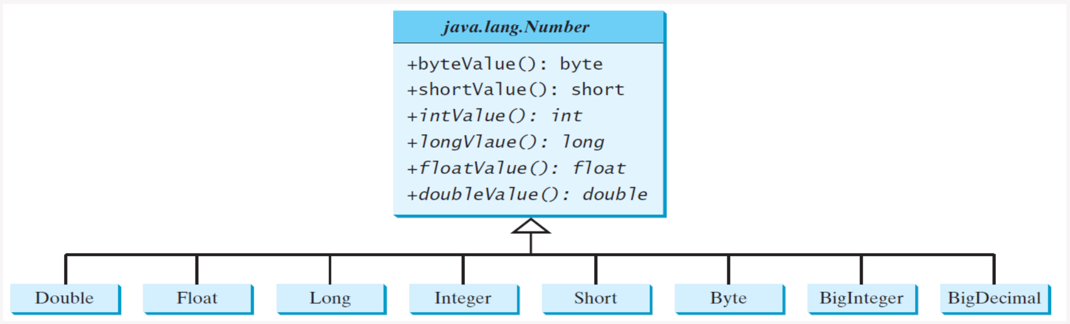
- Superclass of Abstract Class May Be Concrete
  - A subclass can be abstract even if its superclass is concrete.
  - For example, the Object class is concrete, but its subclasses, such as GeometricObject, may be abstract.

- Abstract Class as Type
  - You cannot 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

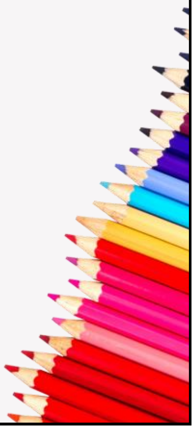
- Number is an abstract superclass for numeric wrapper classes, BigInteger, and BigDecimal.



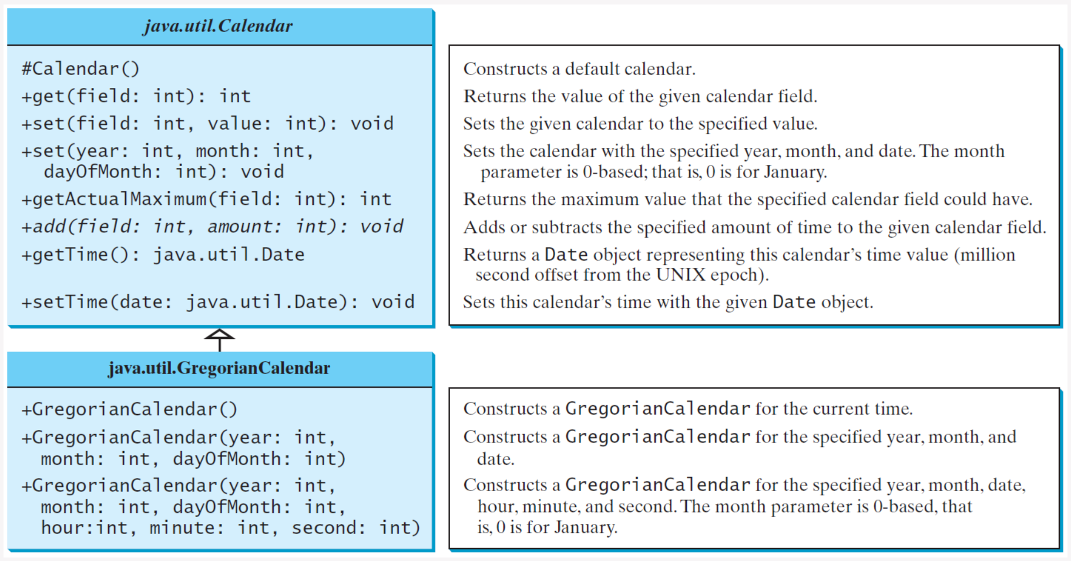
LargestNumbers Run

Case Study: Calendar and GregorianCalendar

- 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 Abstract Calendar Class and Its GregorianCalendar Subclass

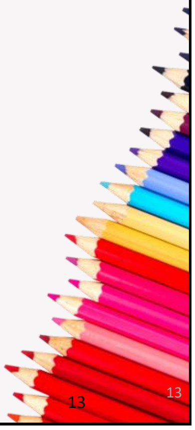


java.util.Calendar	
`#Calendar()`	Constructs a default calendar.
`+get(field: int): int`	Returns the value of the given calendar field.
`+set(field: int, value: int): void`	Sets the given calendar to the specified value.
`+set(year: int, month: int, dayOfMonth: int): void`	Sets the calendar with the 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(field: 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.

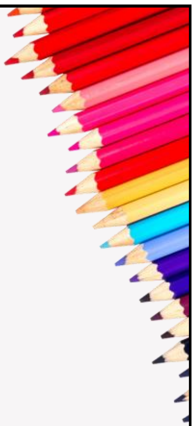
java.util.GregorianCalendar	
`+GregorianCalendar()`	Constructs a `GregorianCalendar` for the current time.
`+GregorianCalendar(year: int, month: int, dayOfMonth: int)`	Constructs a `GregorianCalendar` for the specified year, month, and date.
`+GregorianCalendar(year: int, month: int, dayOfMonth: int, 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 GregorianCalendar Class

- You can use **`new`** `GregorianCalendar()` to construct a default GregorianCalendar with the current time and 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(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.



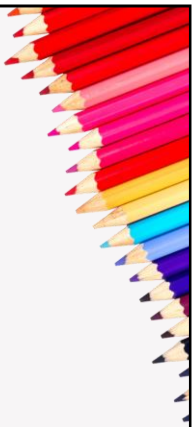
The get Method in Calendar Class



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).

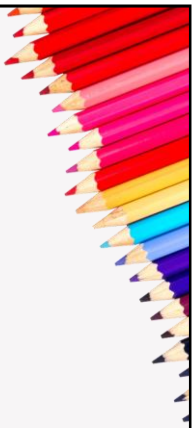
TestCalendar

Run



13.5 Interfaces

- What is an interface?
  - *An interface is a class-like construct that contains only constants and abstract methods.*
- Why is an interface useful?
  - 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:

```
modifier interface InterfaceName {
  /** Constant declarations */
  /** Abstract method signatures */
}
```

- For example,

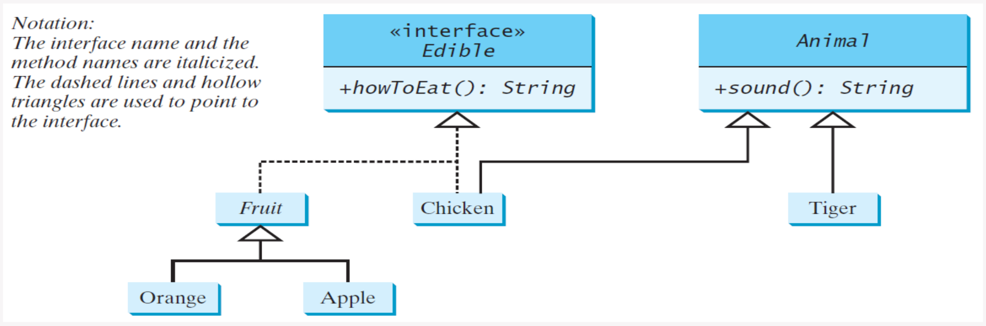
```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

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 a variable, as the result of casting, and so on.

Example

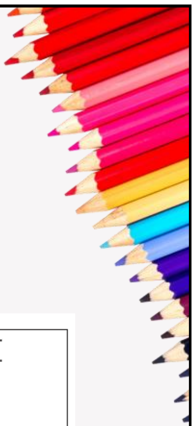
- You can now use the Edible interface to specify whether an object is edible. This is accomplished by letting the class for the object implement this interface using the implements keyword. For example, the classes Chicken and Fruit implement the Edible interface (See TestEdible).


Notation:
The interface name and the method names are italicized. The dashed lines and hollow triangles are used to point to the interface.
«interface» Edible
+howToEat(): String
Animal
+sound(): String
Fruit
Chicken
Tiger
Orange
Apple

Edible

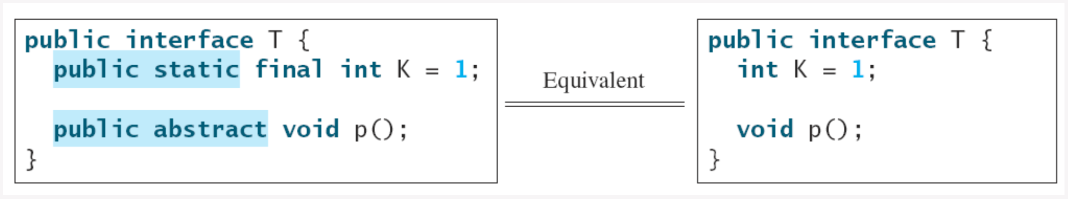
TestEdible

Run



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 T {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

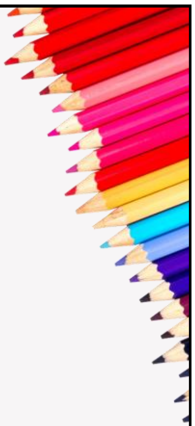
```
public interface T {
  int K = 1;

  void p();
}
```

- A constant defined in an interface can be accessed using syntax *InterfaceName.CONSTANT_NAME* (e.g., T1.K).

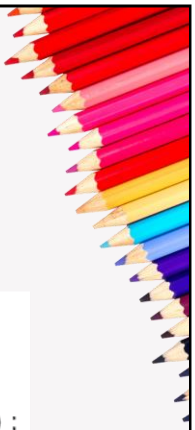
19

19



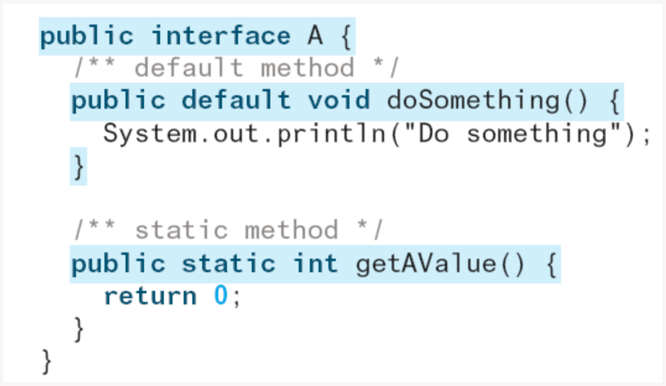
Java 8 Features: default methods

- Java 8 introduced default interface methods using the keyword **default**.
- A default method provides a default implementation for the method in the interface.
- A class that implements the interface may simply use the default implementation for the method or override the method with a new implementation.
- This feature enables you to add a new method to an existing interface with a default implementation without having to rewrite the code for the existing classes that implement this interface.



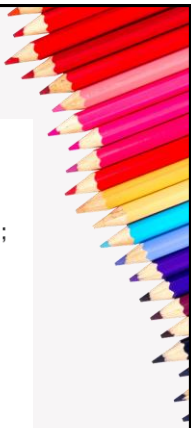
Java 8 Features: static methods

- Java 8 also permits public static methods in an interface.
- A public static method in an interface can be used just like a public static method in a class.



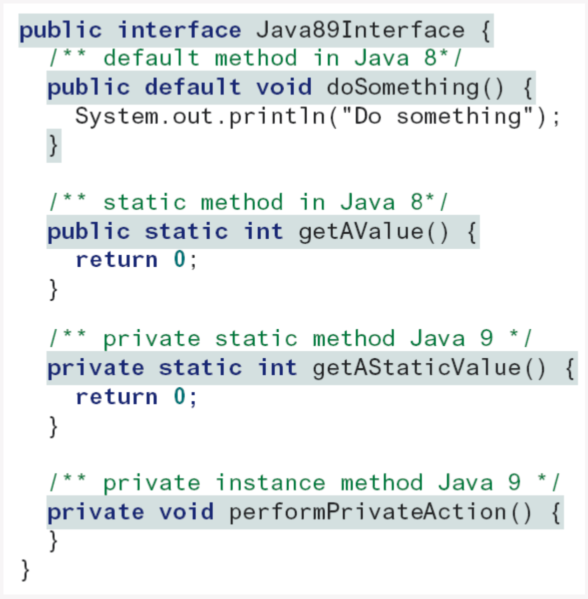
```
public interface A {
   /** default method */
   public default void doSomething() {
      System.out.println("Do something");
   }

   /** static method */
   public static int getAValue() {
      return 0;
   }
}
```



Java 9 Features: private methods

- In Java 9, you can also use private methods in an interface.
- These methods are used for implementing the default methods and public static methods.



```
public interface Java89Interface {
   /** default method in Java 8*/
   public default void doSomething() {
      System.out.println("Do something");
   }

   /** static method in Java 8*/
   public static int getAValue() {
      return 0;
   }

   /** private static method Java 9 */
   private static int getAStaticValue() {
      return 0;
   }

   /** private instance method Java 9 */
   private void performPrivateAction() {
   }
}
```

13.6 The Comparable Interface

- *The **Comparable** interface defines the compareTo method for comparing objects.*
- The interface is defined as follows:

```
// Interface for comparing objects, defined in java.lang
package java.lang;

public interface Comparable<E> {
  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.

Comparable<E>

- The Comparable interface is a generic interface. The generic type E is replaced by a concrete type when implementing this interface.
- Since all the numeric wrapper classes and the `Character` class implement the `Comparable` interface, the `compareTo` method is implemented in these classes.
- Thus, numbers are comparable, strings are comparable, and so are dates. You can use the `compareTo` method to compare two numbers, two strings, and two dates.

Integer ,BigInteger, String and Date Classes

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

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

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

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

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

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

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

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

Comparable Objects

- For example, the following code

```java
System.out.println(new Integer(3).compareTo(new Integer(5)));
System.out.println("ABC".compareTo("ABE"));
java.util.Date date1 = new java.util.Date(2013, 1, 1);
java.util.Date date2 = new java.util.Date(2012, 1, 1);
System.out.println(date1.compareTo(date2));
```

- displays

```
-1
-2
 1
```

Generic `sort` Method

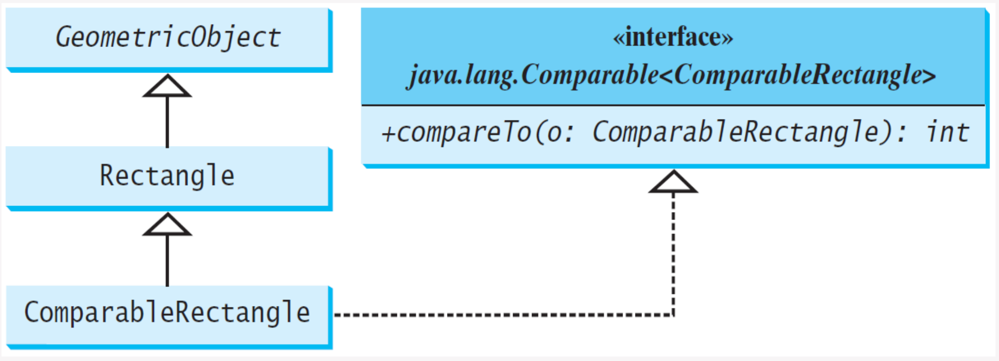
- Let **n** be an Integer object, **s** be a String object, and **d** be a Date object. All the following expressions are **true.**

`n instanceof Integer`	`s instanceof String`	`d instanceof java.util.Date`
`n instanceof Object`	`s instanceof Object`	`d instanceof Object`
`n instanceof Comparable`	`s instanceof Comparable`	`d instanceof Comparable`

- Since all Comparable objects have the compareTo method, the **java.util.Arrays.sort(Object[])** method in the Java API uses the **compareTo** method to compare and sorts the objects in an array, provided that the objects are instances of the Comparable interface.
  - The java.util.Arrays.sort(array) method requires that the elements in an array are instances of Comparable<E>.

SortComparableObjects   Run

Defining Classes to Implement Comparable


GeometricObject
Rectangle
ComparableRectangle
«interface»
java.lang.Comparable<ComparableRectangle>
+compareTo(o: ComparableRectangle): int

ComparableRectangle   SortRectangles   Run

13.7 The Cloneable Interface

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

```
package java.lang;

public interface Cloneable {
}
```

- **Marker Interface**: An empty interface.
- A marker interface does not contain constants or methods. It 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.

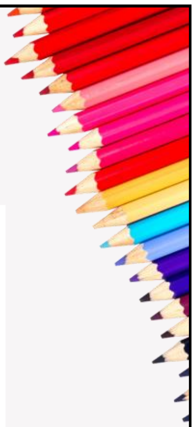
Examples

- Many classes in the Java library (e.g., Date, Calendar, and ArrayList) implement **Cloneable**. Thus, the instances of these classes can be cloned. For example, the following code

```
Calendar calendar = new GregorianCalendar(2013, 2, 1);
Calendar calendar1 = calendar;
Calendar calendar2 = (Calendar)calendar.clone();
System.out.println("calendar == calendar1 is " +
  (calendar == calendar1));
System.out.println("calendar == calendar2 is " +
  (calendar == calendar2));
System.out.println("calendar.equals(calendar2) is " +
  calendar.equals(calendar2));
```

- displays

```
calendar == calendar1 is true
calendar == calendar2 is false
calendar.equals(calendar2) is true
```



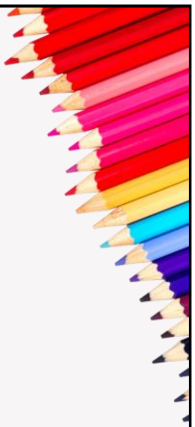
Examples

- The following code:

```
ArrayList<Double> list1 = new ArrayList<>();
list1.add(1.5);
list1.add(2.5);
list1.add(3.5);
ArrayList<Double> list2 = (ArrayList<Double>)list1.clone();
ArrayList<Double> list3 = list1;
list2.add(4.5);
list3.remove(1.5);
System.out.println("list1 is " + list1);
System.out.println("list2 is " + list2);
System.out.println("list3 is " + list3);
```

displays

```
list1 is [2.5, 3.5]
list2 is [1.5, 2.5, 3.5, 4.5]
list3 is [2.5, 3.5]
```



Examples

- You can clone an array using the **clone** method. For example, the following code:

```
int[] list1 = {1, 2};
int[] list2 = list1.clone();
list1[0] = 7;
list2[1] = 8;
System.out.println("list1 is " + list1[0] + ", " + list1[1]);
System.out.println("list2 is " + list2[0] + ", " + list2[1]);
```

displays

```
list1 is 7, 2
list2 is 1, 8
```

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 header for the **clone** method defined in the **Object** class is:

```
protected native Object clone() throws CloneNotSupportedException;
```

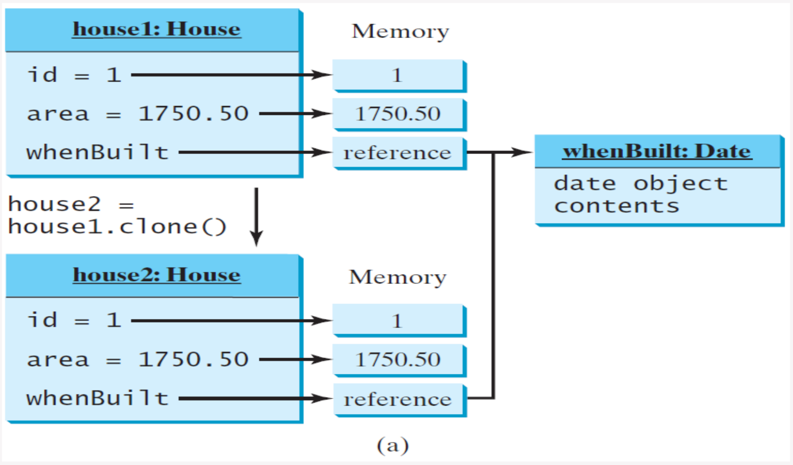
- The following code defines a class named House that implements Cloneable and Comparable.

House

Shallow vs. Deep Copy

```
House house1 = new House(1, 1750.50);
House house2 = (House)house1.clone();
```

Shallow Copy

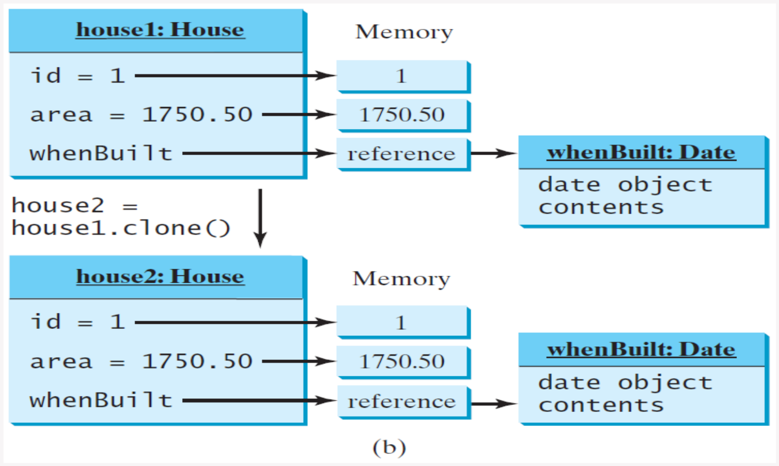


(a)

Shallow vs. Deep Copy

```
House house1 = new House(1, 1750.50);
House house2 = (House)house1.clone();
```

Deep Copy



Shallow Copy

- The default clone method performs a shallow copy.

```
@Override /** Override the protected clone method defined in
  the Object class, and strengthen its accessibility */
public Object clone() throws CloneNotSupportedException {
  return super.clone();
}
```

- CloneNotSupportedException is thrown if House does not implement Cloneable

Deep Copy

- To perform a deep copy for a House object, replace the **clone()** method in lines 26–28 with the following code:

```
public Object clone() throws CloneNotSupportedException {
  // Perform a shallow copy
  House houseClone = (House)super.clone();
  // Deep copy on whenBuilt
  houseClone.whenBuilt = (java.util.Date)(whenBuilt.clone());
  return houseClone;
}
```

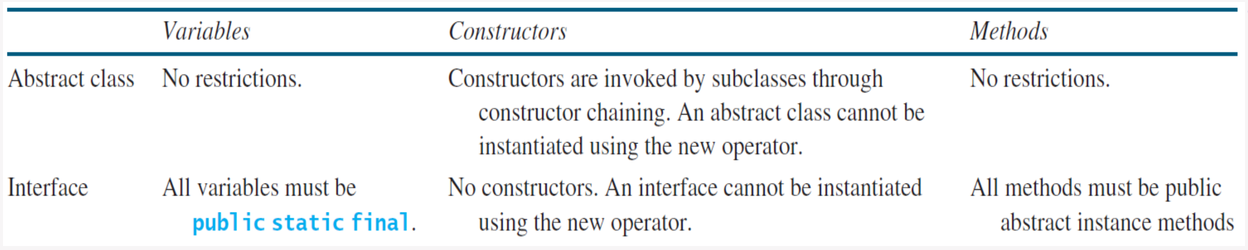
Deep Copy

- or

```
public Object clone() {
  try {
    // Perform a shallow copy
    House houseClone = (House)super.clone();
    // Deep copy on whenBuilt
    houseClone.whenBuilt = (java.util.Date)(whenBuilt.clone());
    return houseClone;
  }
  catch (CloneNotSupportedException ex) {
    return null;
  }
}
```

13.8 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

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

```
public class NewClass extends BaseClass
    implements Interface1, ... , InterfaceN {
  ...
}
```

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

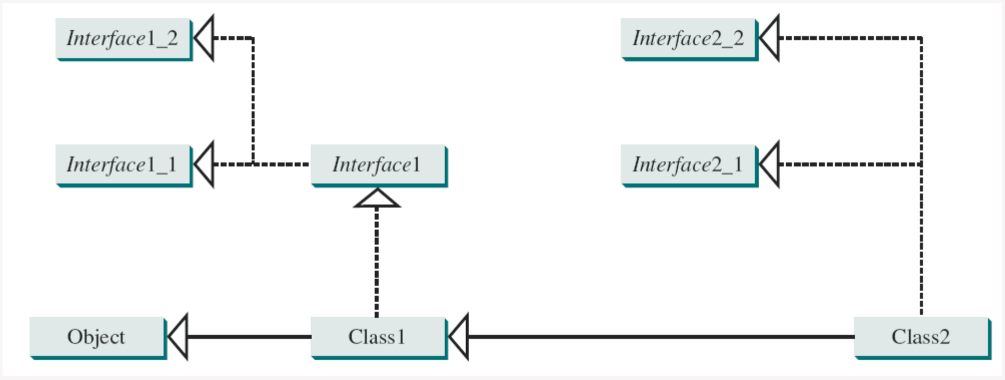
```
public interface NewInterface extends Interface1, ... , InterfaceN {
  // constants and abstract methods
}
```

Interfaces vs. Abstract Classes

- 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.

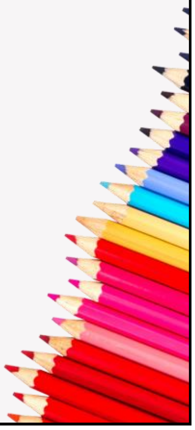
Interfaces vs. Abstract Classes, cont.

- 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.

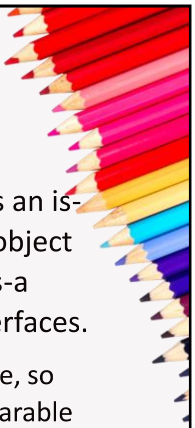


Caution: Conflict Interfaces

- 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.

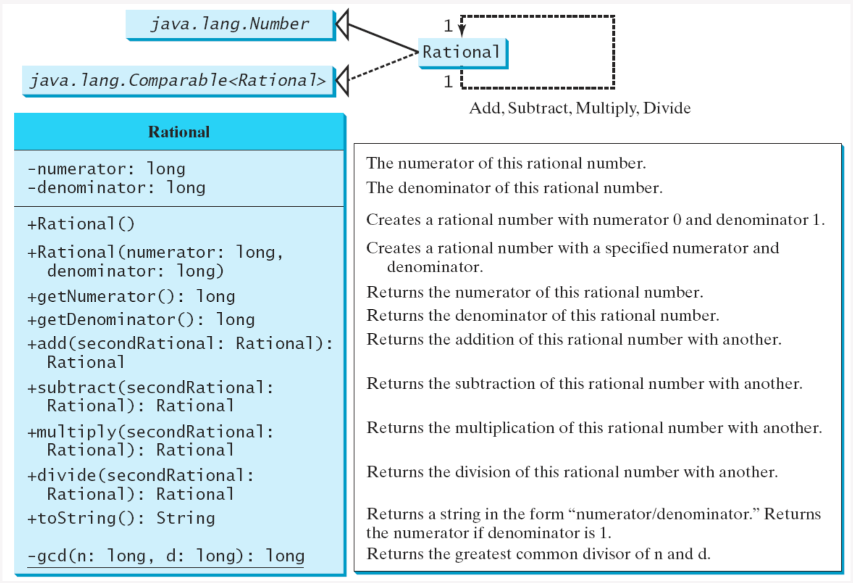


Whether to Use An Interface or A Class?



- 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.
- 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.

The `Rational` Class


java.lang.Number
java.lang.Comparable<Rational>
Rational
1
1
Add, Subtract, Multiply, Divide

Rational	
-numerator: long	The numerator of this rational number.
-denominator: long	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.

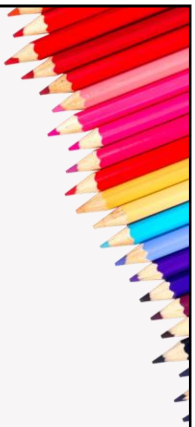
Rational

TestRationalClass

Run

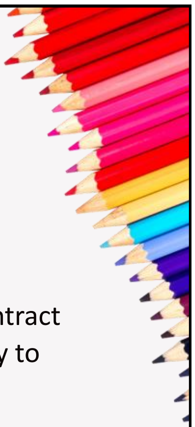
13.10 Class Design Guidelines

- *Cohesion*
  - A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose.
  - You can use a class for students, for example, but you should not combine students and staff in the same class, because students and staff have different entities.
- *Separating responsibilities*
  - A single entity with too many responsibilities can be broken into several classes to separate responsibilities.
  - The classes String, StringBuilder, and StringBuffer all deal with strings, for example, but have different responsibilities. The String class deals with immutable strings, the StringBuilder class is for creating mutable strings, and the StringBuffer class is similar to StringBuilder except that StringBuffer contains synchronized methods for updating strings.



Class Design Guidelines

- Consistency
  - Follow standard Java programming style and naming conventions. Choose informative names for classes, data fields, and methods. A popular style is to place the data declaration before the constructor and place constructors before methods.
  - Make the names consistent.
  - In general, you should consistently provide a public no-arg constructor for constructing a default instance.
  - Override the *equals* method and the *toString* method defined in the Object class whenever possible.



Class Design Guidelines

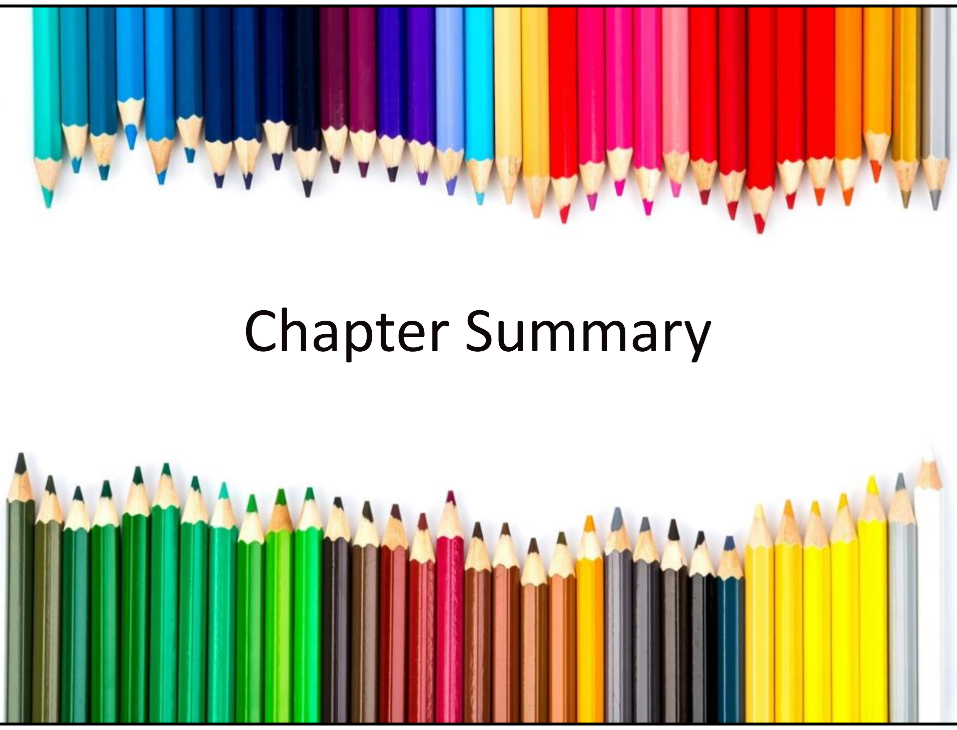
- Encapsulation
  - A class should use the **private** modifier to hide its data from direct access by clients. This makes the class easy to maintain.
  - Provide a getter method only if you want the data field to be readable, and provide a setter method only if you want the data field to be updateable.
- Classes are designed for reuse.
- Clarity
  - A class should have a clear contract that is easy to explain and easy to understand.
  - Methods should be defined intuitively without causing confusion.
  - You should not declare a data field that can be derived from other data fields.

Class Design Guidelines

- Completeness
  - Classes are designed for use by many different customers. In order to be useful in a wide range of applications, a class should 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 variable or method.
  - A variable that is shared by all the instances of a class should be declared static.
- Inheritance vs. Aggregation
  - The difference between inheritance and aggregation is the difference between an is-a and a has-a relationship.

Class Design Guidelines

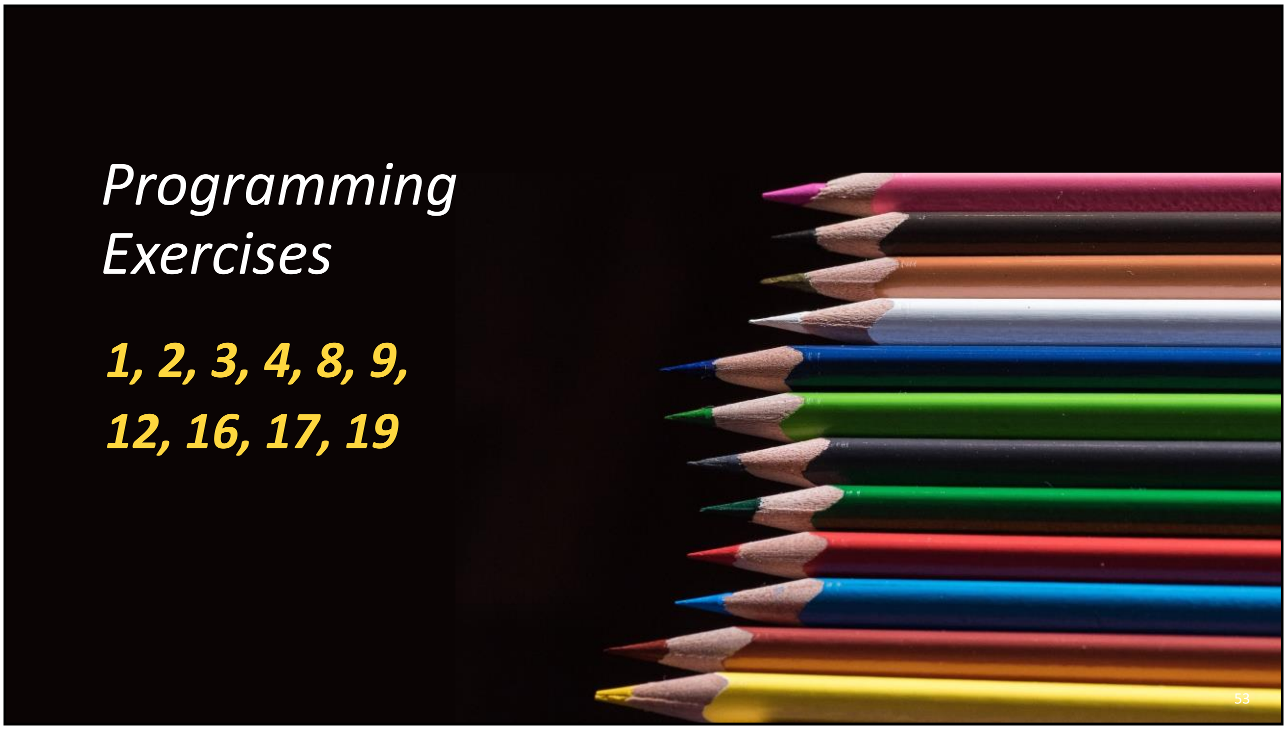
- Interfaces vs. Abstract Classes
  - In general, a strong is-a relationship that clearly describes a parent–child relationship should be modeled using classes.
  - A weak is-a relationship can be modeled using interfaces.
  - Interfaces are more flexible than abstract classes, because a subclass can extend only one superclass but can implement any number of interfaces. However, interfaces cannot contain concrete methods.
  - The virtues of interfaces and abstract classes can be combined by creating an interface with an abstract class that implements it.



Chapter Summary

Chapter Summary

- *Abstract classes are like regular classes with data and methods, but you cannot create* instances of abstract classes using the **new** operator.
- 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 an abstract class can contain constants and abstract methods as well as variables and concrete methods.
- A class can extend only one superclass but can implement one or more interfaces.
- An interface can extend one or more interfaces.



Programming Exercises

1, 2, 3, 4, 8, 9, 12, 16, 17, 19

53