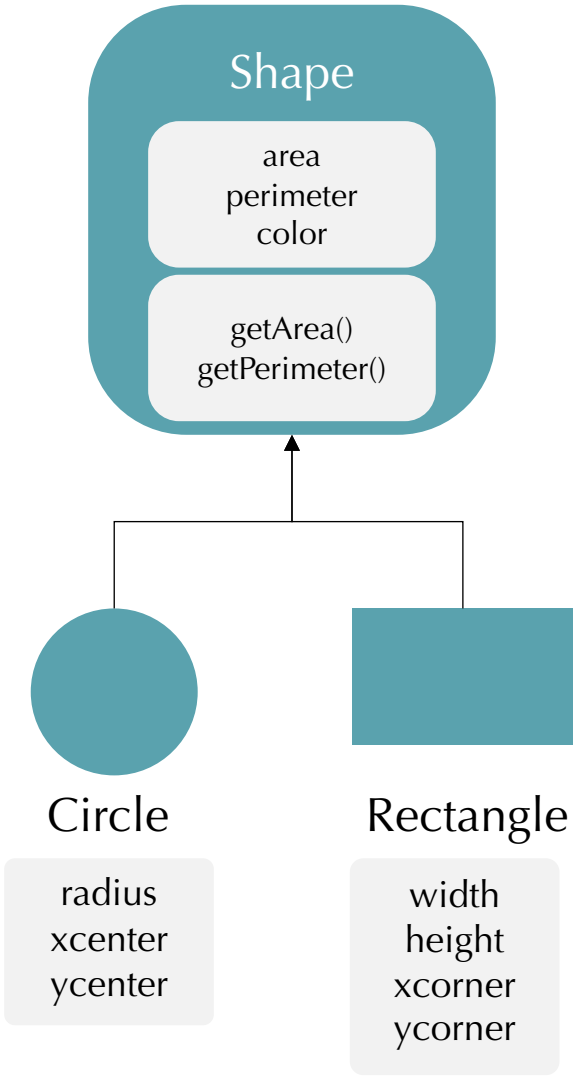
Abstract Classes

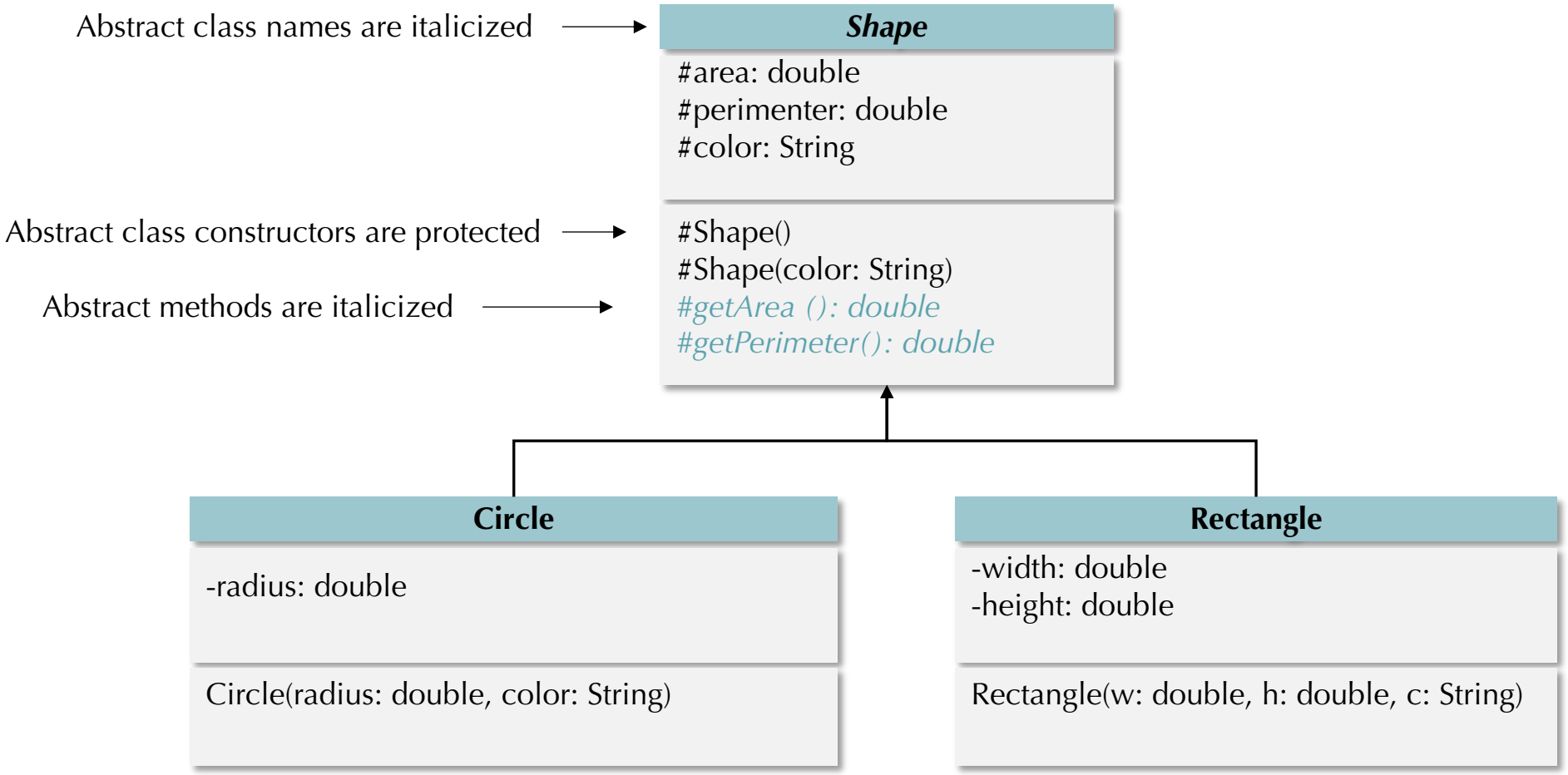
Why Abstract Classes?

- Some classes should not be instantiated such as the Shape class
  - We want Circle and Rectangle objects, not Shape objects
- We need a Shape class, for inheritance and polymorphism
  - We want programmers to instantiate only the subclasses of the Shape

- For example
  - getArea() and getPerimeter() methods are needed for both Circle and Rectangle objects. So, they are defined in the Shape super class
  - However, we cannot implement these methods in the Shape class, i.e., they should only be declared in the Shape without their implementations
  - Therefore, we call them abstract methods



UML Class Diagrams



Abstract Class: Shape

```
// Shape is an abstract class
public abstract class Shape {

  protected String color;
  protected double area;
  protected double perimeter;

  // Constructors of abstract classes should be ‘protected’ to prevent instantiation
  protected Shape() {}

  protected Shape(String color) {
    this.color = color;
  }

  // Abstract classes contain abstract methods without method bodies
  protected abstract double getArea();
  protected abstract double getPerimeter();
}
```

Abstract methods do not have method bodies:
They are implemented in the subclasses.

Derived Classes: Circle and Rectangle

```
public class Circle extends Shape {

  private double radius;

  public Circle(double radius, String color) {
    super(color);
    this.radius = radius;
  }

  @Override
  protected double getArea() {
    area = Math.PI * radius * radius;
    return area;
  }


  @Override
  protected double getPerimeter() {
    perimeter = 2 * Math.PI * radius;
    return perimeter;
  }
}
```

```
public class Rectangle extends Shape {

  private double width;
  private double height;

  public Rectangle(double width, double height, String color) {
    super(color);
    this.width = width;
    this.height = height;
  }

  @Override
  protected double getArea() {
    area = width * height;
    return area;
  }

  @Override
  protected double getPerimeter() {
    perimeter = 2 * (width + height);
    return perimeter;
  }
}
```

Client Code

```
public class AppShape {
  public static void main(String[] args) {

    // Create circle and rectangle objects
    Shape s = new Circle(2,"Red");
    Shape r = new Rectangle(4,6, "Blue");

    // Invoke getArea and getPerimeter methods
    System.out.printf("Area: %5.2f, Perimeter: %5.2f\n", s.getArea(), s.getPerimeter());
    System.out.printf("Area: %5.2f, Perimeter: %5.2f\n", r.getArea(), r.getPerimeter());
  }
}
```

Client Code

```
public class AppShape {
  public static void main(String[] args) {

    // Create circle and rectangle objects
    Shape s = new Circle(2,"Red");
    Shape r = new Rectangle(4,6, "Blue");

    // Check equality of the areas
    System.out.println(equalArea(s, r));
  }

  // Returns true if the areas of two Shape object are equal
  public static boolean equalArea(Shape a, Shape b) {
    return a.getArea() == b.getArea();
  }

}
```

Client Code

```
public class AppShape {
  public static void main(String[] args) {

    Shape s = new Circle(2,"Red");
    Shape r = new Rectangle(4,6, "Blue");

    // Print information about objects
    printInfo(s);
    printInfo(r);

  }

  public static void printInfo(Shape inputShape) {
    if (inputShape instanceof Circle)
      System.out.println("Shape is a circle");
    else if (inputShape instanceof Rectangle)
      System.out.println("Shape is a rectangle");
    else
      System.out.println("Another shape.");
  }
}
```

Abstract Classes

- A class that contains abstract methods must be defined as abstract
- An abstract method is defined without implementation: Its implementation is provided by the subclasses
- Abstract classes are like regular classes, but you cannot create instances of using the new operator:

```
Person a = new Person(); // This is wrong if Person class is abstract
```

- The constructor in the abstract class is protected, because it is used only by subclasses
- Abstract class can be used as a data type (polymorphism)

```
Shape a = new Circle();
Shape[] shapes = new Shape[10];
shapes[0] = new Rectangle();
```

- A subclass can be abstract even if its superclass is concrete

Notes on 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.
  - In other words, in a nonabstract subclass extended from an abstract class, all the abstract methods must be implemented.
  - Abstract methods are nonstatic

Notes on Abstract Classes

- 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

Notes on Abstract Classes

- 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

Notes on Abstract Classes

- 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
  - For example, the Java's Object class is concrete, but its subclasses, such as Shape, may be abstract

Interfaces

Interfaces

- Interface's intent is to specify common behavior for objects
  - It is a collection of method declarations with no bodies
- Methods of an interface are always empty; they are simply method signatures
- Interfaces do not have constructors and they cannot be directly instantiated
- When a class implements an interface, it must implement all of the methods declared in the interface
- Interfaces practically enforces application programming interface (API)

Declaring Interfaces

Sellable is an interface

```
public interface Sellable {

  // returns the price
  public int listPrice();

  // returns the lowest possible price
  public int lowestPrice();

  // reduces the price
  public void makeDiscount();

}
```

Product class implements the Sellable interface:
It should implement all the methods in the interface

```
public class Product implements Sellable {

  private int price;
  private double discountRate = 0.1;

  @Override
  public int listPrice() {
    return price;
  }

  @Override
  public int lowestPrice() {
    return price / 2;
  }

  @Override
  public void makeDiscount() {
    price = (int) (price * discountRate);
  }
}
```

Multiple Interfaces

- In Java, a class can implement multiple interfaces (it may only extend one other class)

Product class implements multiple interfaces

```
public class Product implements Sellable, Printable {

}
```

A subclass may implement an interface

```
public class Circle extends Shape implements Printable {

}
```

Multiple Inheritance for Interfaces

- The ability of extending from more than one type is known as multiple inheritance
- In Java, multiple inheritance is allowed for interfaces but not for classes
- We can use multiple inheritance for “mixing” the methods from two or more interfaces to define an interface that combines their functionality (possibly adding more methods)

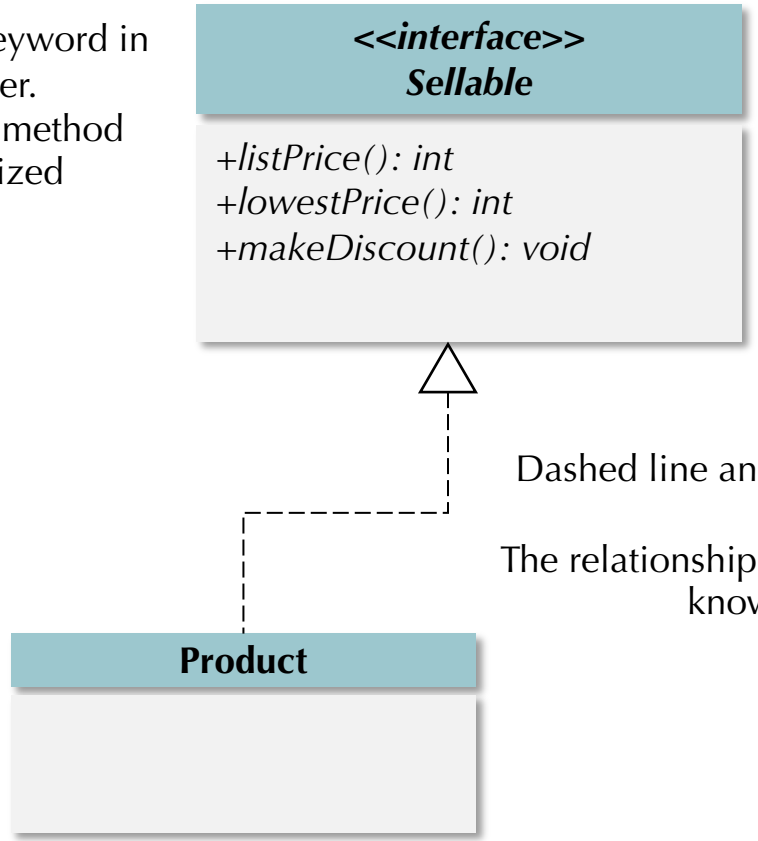
```
public interface A extends B, C {

  public int myMethod();

}
```

UML Diagrams for Interfaces

Use <<interface>> keyword in the UML header.
Interface name and method names are italicized



Dashed line and hollow triangle are used to point to the interface.
The relationship between the class and the interface is known as interface inheritance

Interfaces

- You can use an interface as a data type for a reference variable

```
Sellable myItem = new Product();
```

- Like abstract classes, you cannot create an instance from an interface

```
Sellable myItem = new Sellable(); // Wrong
```

Interfaces

- In an interface, all methods are public and abstract
- In an interface, you can include data fields. All data fields are public static final
- Java allows these modifiers to be omitted. Therefore, the following interface definitions are equivalent

```
public interface Sellable {

  public static final int a = 2;

  public abstract int listPrice();

  public void makeDiscount();

}
```

```
public interface Sellable {

  int a = 2;

  int listPrice();

  void makeDiscount();

}
```

Interfaces vs Abstract Classes

	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.	May contain public abstract instance methods, public default, and public static methods.

Interface Example

Comparable Interface

Comparable Interface

- Suppose you want to design a generic method to find the larger of two objects of the same type, such as two students, two circles, etc.
- In order to accomplish this, the two objects must be comparable
- Java provides the Comparable interface for this purpose.
- Comparable interface specifies only one method to be implemented: compareTo()

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

- The class, such as the Circle class, should implement this method

Comparable Interface

- 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

- For example, String are comparable since the String class implements the compareTo() method:

```
String a = "AAA";
String b = "CCC";
System.out.println(a.compareTo(b)); // outputs -2 since AAA is less than CCC
System.out.println(b.compareTo(a)); // outputs 2
```

Comparable Interface

- Arrays.sort() method in the Java API uses the compareTo method to compare and sorts the objects in an array, provided the objects are instances of the Comparable interface

```
String[] cities = {"New York", "Barcelona", "Istanbul"};
Arrays.sort(cities);   // sort the string array
System.out.println(Arrays.toString(cities)); // prints Barcelona, Istanbul, New York
```

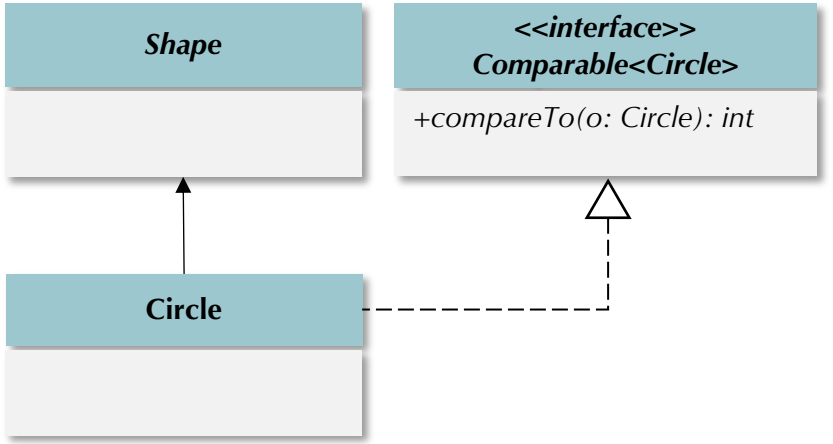
Comparable Interface

- Write a comparable Circle class, inherited from the Shape class

```
public class Circle extends Shape implements Comparable<Circle>{

  // data fields, constructors, and other methods are here

  @Override
  public int compareTo(Circle o) {
    if (getArea() > o.getArea())
      return 1;    // if the current object is greater than o,
                   // return positive number
    else if (getArea() < o.getArea()) // if the current object is less than o,
                                      // return negative
      return -1;
    else // if they are equal, return zero
      return 0;
  }
}
```



Comparable Interface

- Test the comparable Circle class

```
Circle c1 = new Circle(2,"Red");
Circle c2 = new Circle(5,"Blue");
System.out.println(c1.compareTo(c2)); // prints -1 since c1 is less (smaller) than c2



// Example: Sort circles based on their areas
Circle[] circles = new Circle[3];
circles[0] = new Circle(3,"Yellow");
circles[1] = new Circle(7,"Black");
circles[2] = new Circle(1,"White");
Arrays.sort(circles);
System.out.println(Arrays.toString(circles)); // prints the circles 1, 3, and 7 in increasing order
```