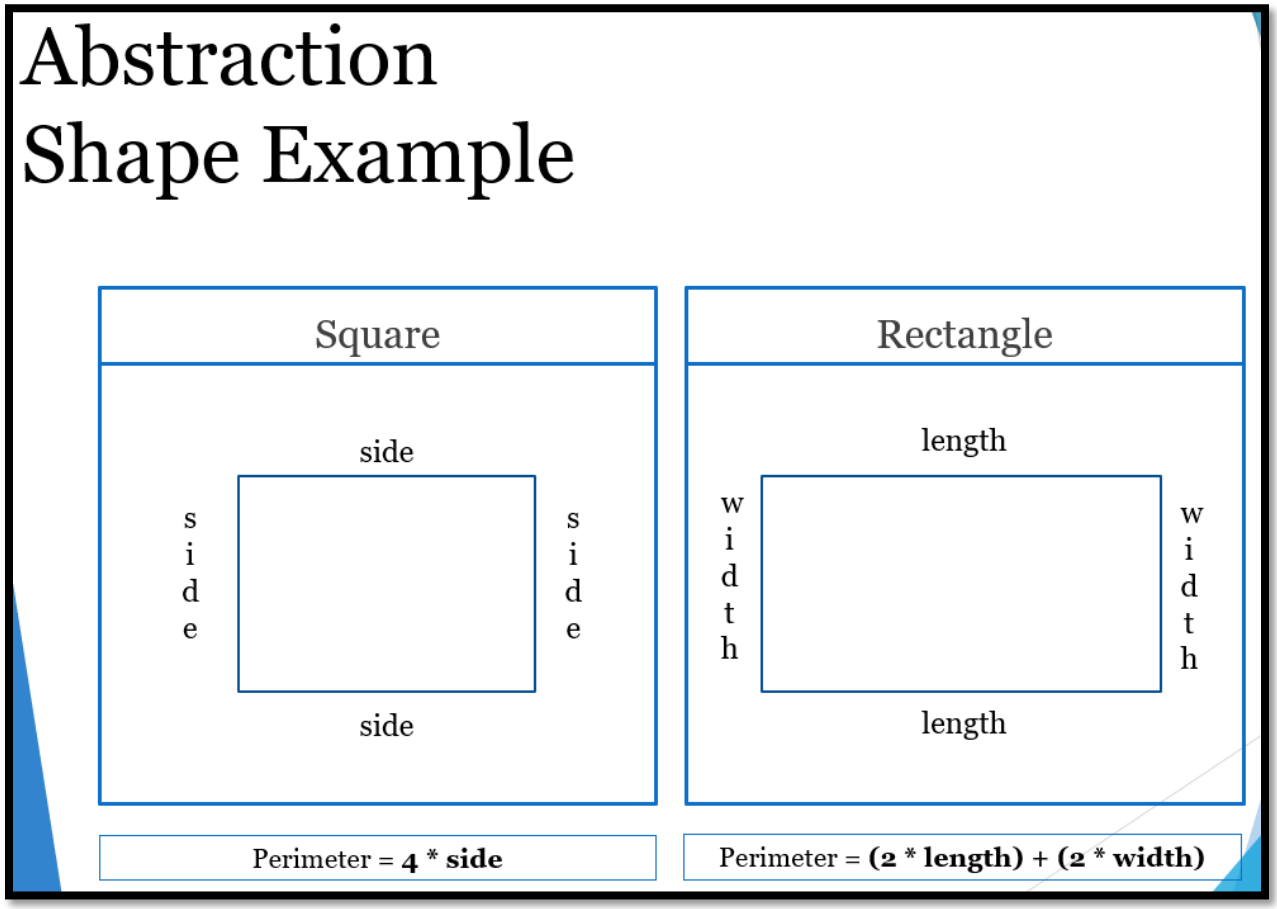
Rex Jones II

(Transcript) Abstraction Classes & Methods

Abstract Classes & Methods

An abstract class is a class with empty methods and optional concrete methods. The empty method is an abstract method and the concrete method is an instance method with fully defined details. In this example, we see 2 diagrams: a square and a rectangle. To calculate the perimeter, we can define one generic abstract method in the superclass then let the subclass for square and subclass for rectangle provide details on how to calculate the perimeter. The calculations for square are 4 times the side and rectangle is 2 times length plus 2 times width. It's a difference because the square is the same on each side but rectangle has a longer length than width.



Rex Jones II

Let's go to Eclipse and create a class and method. The class will be public abstract class Shape and method is public int calculatePerimeter (); Hover over the compile time error and the message states "This method requires a body instead of a semicolon". We have a semicolon because the plan is to not add a body. Now, they offer us 2 quick fixes. We can either add a body or Change Shape.calculatePerimeter to abstract. Add abstract before the return type and the error goes away. Let's see what happens when removing abstract from the class.

Now there are 2 errors, one at the class level and another one at the method level. Hover the method and we see "The abstract method calculatePerimeter in type Shape can only be defined by an abstract class". If we have one or more abstract methods then we must make the class abstract. That's why we see the following quick fixes: Remove abstract modifier and Make type Shape abstract. Hover the class and we see "The type Shape must be an abstract class to define abstract methods" with 1 quick fix Make type Shape abstract. Let's re-add abstract. Finally, the errors go away.

Recall an abstract class is a class with an empty method which means the method is abstract just like calculatePerimeter and it is not required to have a defined method meaning the method has a body. Let's add a defined method which is an instance method

```
public void printShapeMessage () {
  System.out.println("Print Perimeter \n");
}
```

```
public abstract class Shape {

   public abstract int calculatePerimeter ();

   public void printShapeMessage () {
      System.out.println("Print Perimeter \n");
   }
}
```

Next, we are going to create 2 subclasses that will provide details for this calculatePerimeter method. Starting with the Square class, it extends the Shape superclass. Instantly, we see an error "The type Square must implement the inherited abstract method Shape.calculatePerimeter()" with 2 quick fixes. Add the unimplemented method or Make type Square abstract. If we are not going to implement the abstract method then we must make the Square class abstract because the superclass contains an abstract method. Select add unimplemented methods and we see the Override annotation which indicates the calculatePerimeter method is overridden. Remove the comment then implement the details. private int side = 10. Method details are int perimeter = 4 * side; / sysout("Square Perimeter = " + perimeter) / return perimeter;

Rex Jones II

```
public class Square extends Shape {

   private int side = 10;

   @Override
   public int calculatePerimeter() {

      int perimeter = 4 * side;
      System.out.println("Square Perimeter = " + perimeter);
      return perimeter;
   }
}
```

The Rectangle class is similar to the Square class but has different details for the calculatePerimeter method. Rectangle extends Shape
private int length = 5; / private int width = 10; / @Override /
public int calculatePerimeter () {
int perimeter = (2 * length) + (2 * width);
System.out.println("Rectangle Perimeter = " + perimeter)
return perimeter;

```
public class Rectangle extends Shape {

   private int length = 5;
   private int width = 10;

   @Override
   public int calculatePerimeter () {
      int perimeter = (2 * length) + (2 * width);
      System.out.println("Rectangle Perimeter = " + perimeter);
      return perimeter;
   }
}
```

If necessary, you can also create a constructor in an abstract class. That's it for creating an abstract class and abstract methods in a superclass with a subclass implementing the details. Now, let's test out this code in the ShapeTest class. Write main CTRL + SPACE for the shortcut.

With abstraction, we cannot create an object of abstract classes. Our abstract class is Shape, let's add shape as an object = new Shape ();. A compiler error states "Cannot instantiate the type Shape". Comment this line // and say Abstract Classes Cannot Create An Object. There are no objects of an abstract class because abstract classes do not define a complete implementation. However, we will not get an error by declaring a reference of the abstract class Shape shape1 = new Square (); or Shape shape2 = new Rectangle (); These 2 syntaxes are good because the Square type and Rectangle type are not abstract classes. We can also declare a reference of the child class

Square square = new Square ();
square. and we see calculatePerimeter from the Square class and printShapeMessage from the Shape class. Notice we do not see the calculatePerimeter from the Shape class.

Select both methods printShapeMessage() and square.calculatePerimeter().

Rectangle rectangle = new Rectangle();
rectangle.calculatePerimeter();

```
public class ShapeTest {

    public static void main(String[] args) {
        //Shape shape = new Shape (); Abstract Classes Cannot Create An Object
        Shape shape1 = new Square ();
        Shape shape2 = new Rectangle ();

        Square square = new Square ();
        square.printShapeMessage();
        square.calculatePerimeter();

        Rectangle rectangle = new Rectangle ();
        rectangle.calculatePerimeter();
    }
}
```

It's not required to include the instance method which provide a default action. Let's run. The console shows Print Perimeter / Square Perimeter = 40 and Rectangle Perimeter = 30. Next, we are going to take a look at Interfaces.