Rex
Jones II

(Transcript) Java
Object-Oriented Programming

Table of Contents

Rex
Jones II

Java OOP Video Playlist

https://www.youtube.com/playlist?list=PLfp-cJ6BH8u9QG45AI03X9iYxYU_UHL0Y

Introduction

Hello and Welcome, in this series, let's talk about Object-Oriented Programming better known as OOP. In this Intro To OOP, we are going to discuss Structured Programming and Object-Oriented Programming.

OOP took the best ideas from Structured Programming and combined those ideas with a few new concepts. We have the option of organizing our program around code or we can organize our program around data. Structured Programming is organized around code while the OOP's concept is organized around data.

My name is Rex and I like to share programming and automation knowledge. If you are interested, connect with me on YouTube, Twitter, LinkedIn, Facebook, and GitHub. All of the documents including the transcript will be placed on GitHub. Videos are released 3 times a week on Monday, Wednesday, and Friday.

Slide 3

When it comes to Structured Programming, the programmer divides the whole program into smaller units. For example, let's take an employee working at a company. First, they submit their resume, Attend the interview or interviews, Get hired at the company, Perform their job responsibilities, then Receive a paycheck. The big program is divided into small units such as Calculate Salary, Calculate Bonus, and Calculate Health Benefits. That's what we call code acting on data.

Slide 4

Object-Oriented Programming is different. With this same example of an employee working at a company. Our program has a class called Employee. It's organized around data and the data controls access to the code. We define the data and we define the methods that are allowed to act on the data. The data is age, employeeID, and title while the methods are calculateSalary and calculateBonus.

Slide 5

Here's a side by side comparison of Structured and Object-Oriented Programming. There was a point when Structured Programming reached its limit and could not handle complex applications. Object-Oriented Programming was created to help developers deal with more and more and more and more challenges as the software application grew.

Slide 6

The OOP's concept deals with complex application using Classes and Objects. Our class is Employee and our objects are jane and john. Classes are the foundation which builds the entire Java programming language. Also, classes define the nature of an object. In other words, a class is a copy or template for an object. The same way class has age, our object jane has age and the value is 25. Just like the Employee class has employeeID and title. Jane has employeeID with a value of 34 and Automation Engineer as the title. The same goes for each method calculateSalary and calculateBonus. Jane has access to both methods calculateSalary and calculateBonus. We see how a class serves as a pattern for creating the object. In the same manner, our object john has access to the same data and methods.

Slide 7

All Object-Oriented Programming languages have 4 common traits. Those 4 traits are Encapsulation, Inheritance, Polymorphism, and Abstraction. In some cases, you might see some people say all OOP languages have 3 common traits: Encapsulation, Inheritance, and Polymorphism. Abstract Classes, Abstract Methods, and Interfaces falling under Inheritance. The concepts remain the same so it does not matter where we place them, as long as we understand the concepts.

- Encapsulation is a way to protect the data by keeping it safe from outside classes. It can only be accessed through a method within its own class
- Inheritance allows a hierarchy creation of classes. It’s a component where 1 class is an extension of a different class. The class that is extended is called a superclass and the class that extends is called a subclass. However, I prefer to say parent class and child class. A child class receives the data and methods from the parent class. This incorporation happens in the declaration of a child class.
- Polymorphism means many forms. It allows an object to acquire many representations. All of those representations operate similar to each other. As a result, 1 interface has the ability to take on properties of more than 1 class.
- Abstraction is when the necessary details are made available. Abstract classes create a parent class that specifies only a general form shared by the child class. An abstract method contains no body and not implemented by the parent class.

That’s it for the Intro To OOP’s. Next, I will demo Encapsulation.

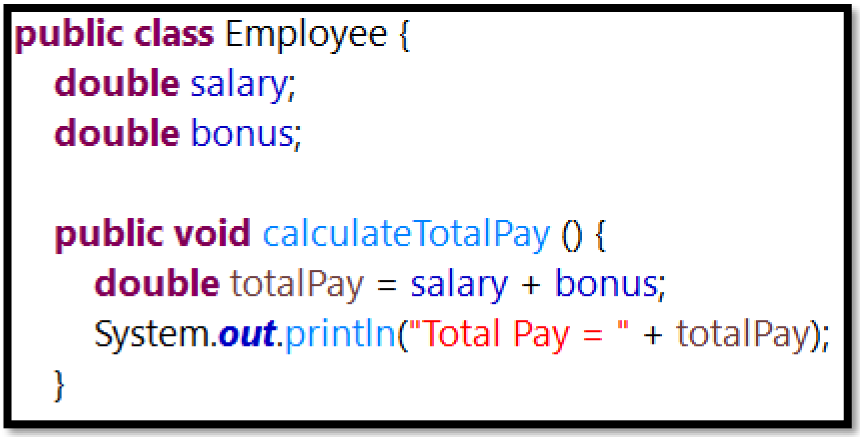
Encapsulation

Encapsulation is the process of protecting the internal representation of an object. It’s known as data hiding because we make the data private so the data cannot be accessed from outside the class.

Hi my name is Rex and I like to share programming and automation knowledge through books, blogs and videos. If you are interested in the content, feel free to connect with me on LinkedIn, YouTube, Twitter, Facebook, and GitHub.

Let’s go to Eclipse.

We have a class called Employee with a double data type for salary and bonus. The method calculateTotalPay performs an action on the data by adding salary and bonus then assigning the value to totalPay.



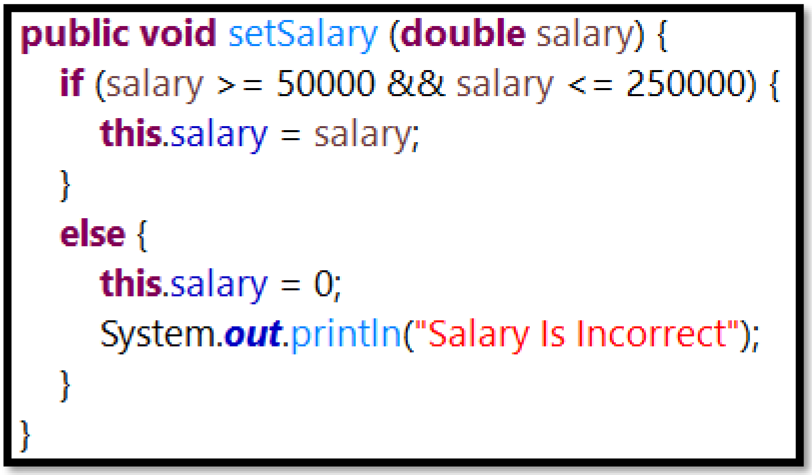
```
public class Employee {
  double salary;
  double bonus;

  public void calculateTotalPay () {
    double totalPay = salary + bonus;
    System.out.println("Total Pay = " + totalPay);
  }
```

Let’s look at why we need to protect our data. In this class called TestEmployeeObject, Jane’s salary should be set to $100,000 and bonus set to $10,000. However, check out what happens when we calculate total pay. Jane made over $1,000,000 with a total pay of $1,010,000. The salary is not correct at $1,000,000. This is a prime example on why we need to protect our data.

A person can make a mistake and enter the wrong data. So, we protect the data using a private access modifier with getter and setter methods. A getter method is used to get and view values while a setter method is used to set or modify a value. Let’s say the company’s policy is no employee gets paid less than $50,000 and no employee gets paid more than $250,000.

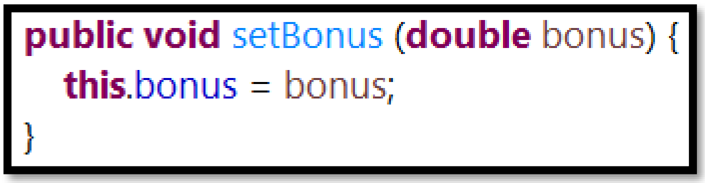
We protect the data by changing salary and bonus to private then Save. Do you see how instantly jane.salary and jane.bonus both have an error? They have an error because they are no longer visible. Next, we create a setter method for salary. Let’s start with salary:

We write public void setSalary. The parameter will be (double salary) inside the parenthesis.
if salary is greater than or equal to $50,000 and salary is less than or equal to $250,000
this.salary equals salary
else this.salary = $0. Let’s add a print statement: sysout (“Salary Is Incorrect”)



```
public void setSalary (double salary) {
  if (salary >= 50000 && salary <= 250000) {
    this.salary = salary;
  }
  else {
    this.salary = 0;
    System.out.println("Salary Is Incorrect");
  }
}
```

The keyword this for this.salary refers to the field private double salary and not the parameter salary in parenthesis. Now, we have a boundary for setting our salary. For bonus, there is no limit so I’m going to create a setter method by writing
public void setBonus (double bonus) as the parameter
this.bonus = bonus



```
public void setBonus (double bonus) {
    this.bonus = bonus;
}
```

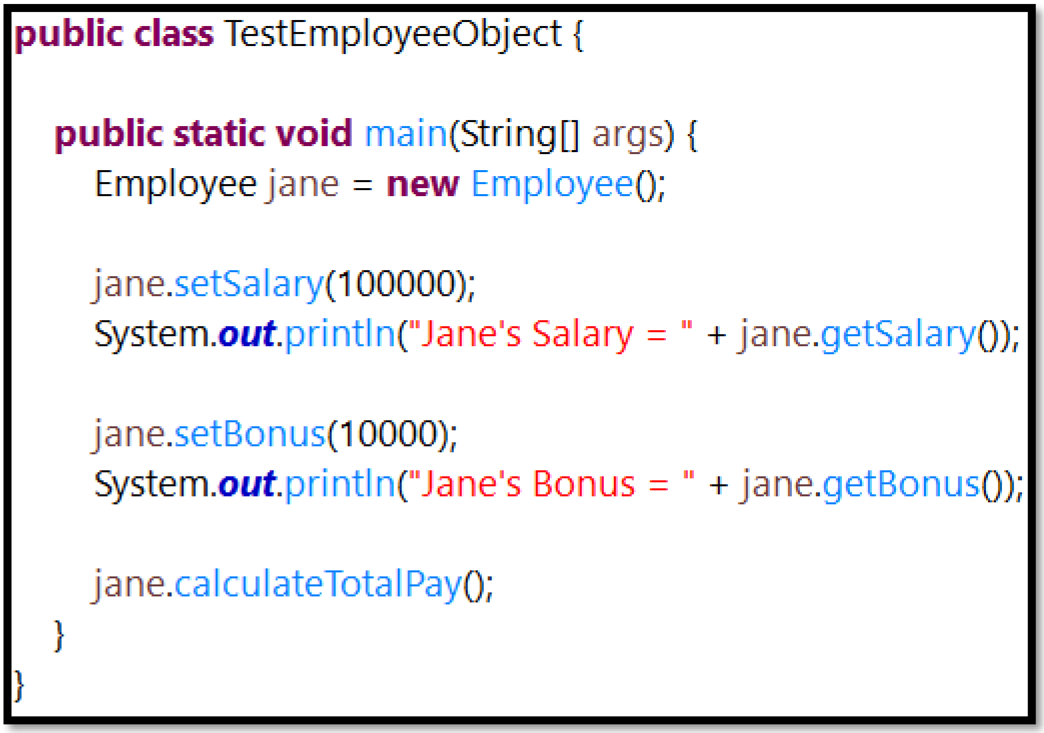
In the TestEmployeeObject class, we must call the methods by writing jane.set. The intellisense shows setBonus and setSalary: select setSalary then pass $1,000,000 again as an argument. Also write jane.setBonus then pass $10,000.

Let’s Run. We protected salary so the console shows Salary Is Incorrect. Change $1,000,000 to $100,000 and run. Now, salary is correct. The Console shows Total Pay equals $110,000. How can we print the salary and/or bonus from the TestEmployeeObject class?

sysout (“Jane’s Salary = “ + jane.) We know the data is not available because it’s private. That’s where the concept of getter methods come into action. It will help us get the data.

We write public. Since the data type is double for salary then the return type is double.
getSalary as the method name no parameters and we are going to return salary }. From the beginning, if you did not want to write each getter and setter method then Eclipse can generate both methods for us. Right click > Select Source > then Generate Getters and Setters then check bonus and click OK. There it is: public double getBonus and return bonus. Now, let’s finish our print statements.

jane.getSalary and sysout (“Jane’s Bonus = ” + jane.getBonus).



```java
public class TestEmployeeObject {

    public static void main(String[] args) {
        Employee jane = new Employee();

        jane.setSalary(100000);
        System.out.println("Jane's Salary = " + jane.getSalary());

        jane.setBonus(10000);
        System.out.println("Jane's Bonus = " + jane.getBonus());

        jane.calculateTotalPay();
    }
}
```

Let me go Save over here. Let's Run. Jane's Salary is $100,000, Bonus is $10,000, and Total Pay is $110,000. Bingo, next we are going to look at Inheritance.

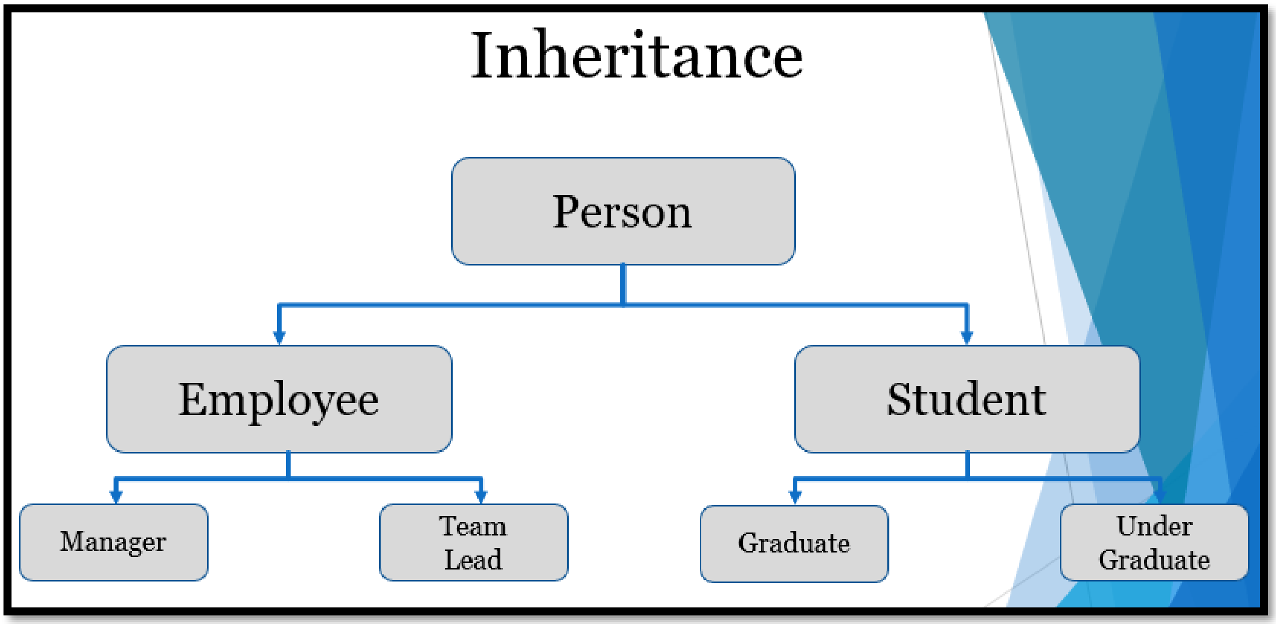
Inheritance

Inheritance is when a class receives the members of another class. It allows us to extend a class then reuse the code. We create a generic class and specialized class. The generic class defines common traits that will be inherited by a specialized class. In Java, the inherited class is called a superclass also known as parent class. The special class which receives an inheritance is called a subclass also known as child class.

Hi my name is Rex and I share automation and programming knowledge. In this video and the next few videos, I am going to share knowledge on Java programming. If you are looking to learn or refresh your knowledge, then connect with me on LinkedIn, YouTube, Twitter, Facebook and GitHub. You can download the code and each transcript from GitHub.

With Object-Oriented Programming, Inheritance is known for representing an **"is a"** relationship. For example, an Employee **is a** Person, so the Employee class can be a child class of the Person class. In the same way, a Student **is a** Person and Student can be a child class of the Person class. With this hierarchy, the Person class is the parent of 2 child classes: Employee and Student. The child classes can become a parent to other classes.

When it comes to sharing data, both child classes and each grandchild inherit from the Person parent class. Let's say, the Person class has a variable called name and a method called getName. From left to right, the Employee, Student, Manager, Team Lead, Graduate, and Undergraduate classes inherit those same members from the Person class.



Inheritance

Person

Employee

Student

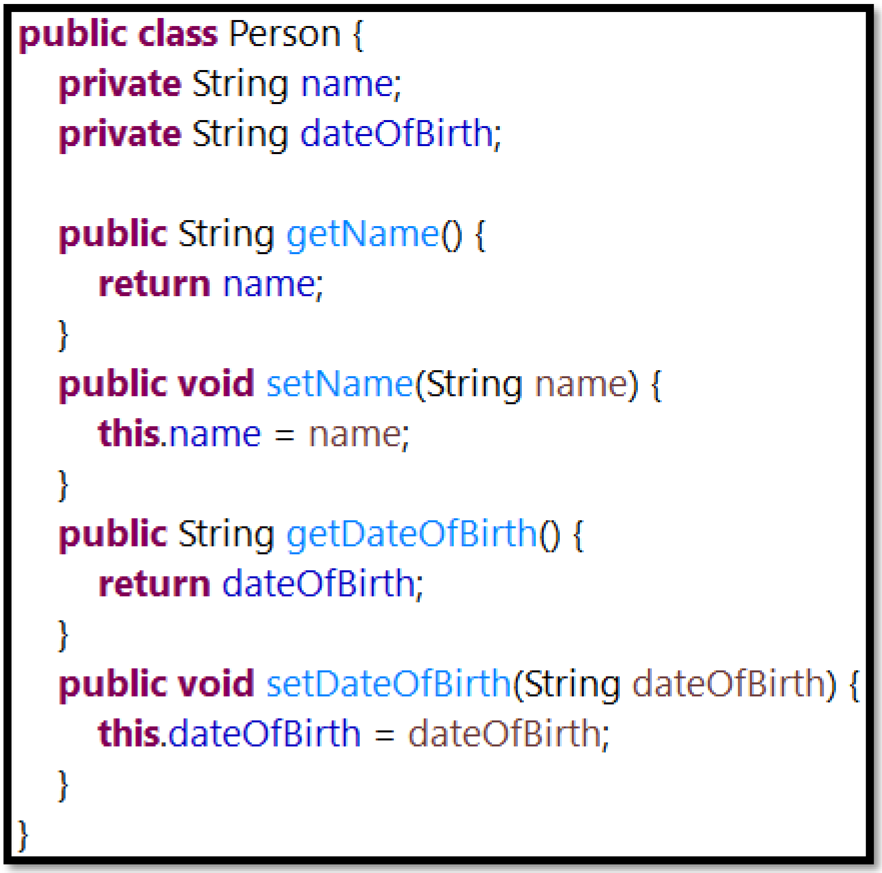
Manager

Team Lead

Graduate

Under Graduate

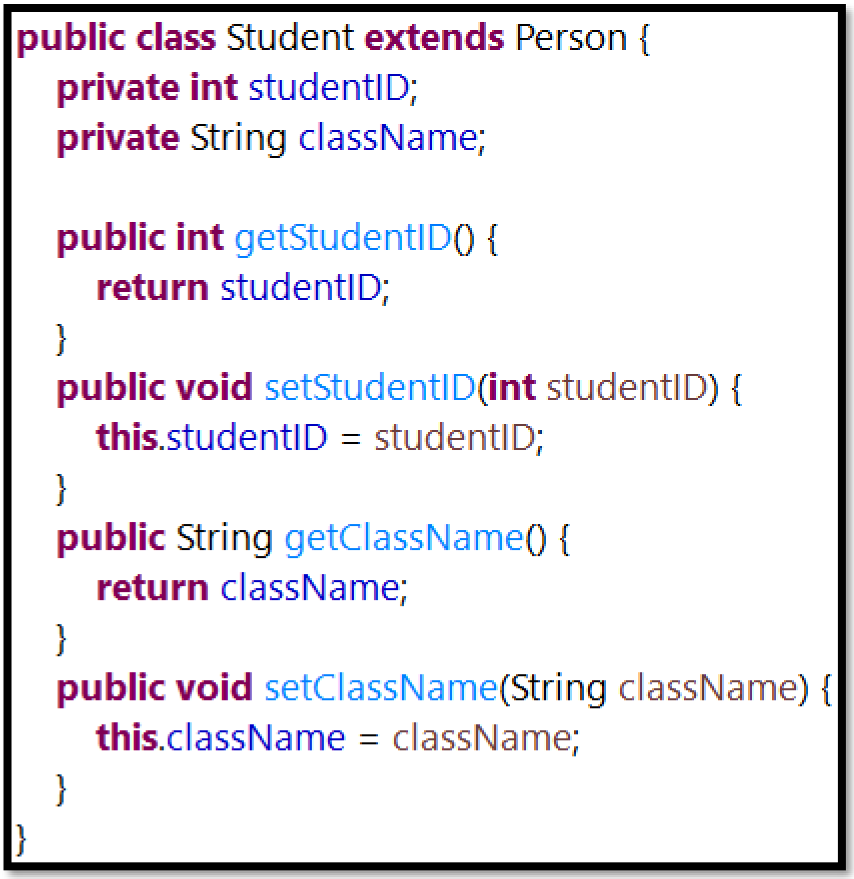
In Eclipse, we are going to write 2 variables for the Person class.
private String name and private String dateOfBirth. Next, let’s add the getter and setter methods by right clicking > Source > Generate Getters and Setters > expand both checkboxes. Eclipse will automatically add these 4 getter and setter methods. Check dateOfBirth and name then click OK. We have all 4 methods.



```
public class Person {
   private String name;
   private String dateOfBirth;

   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public String getDateOfBirth() {
      return dateOfBirth;
   }
   public void setDateOfBirth(String dateOfBirth) {
      this.dateOfBirth = dateOfBirth;
   }
}
```

Next is the Student child class. The purpose of Inheritance is to reuse code by extending a class. Therefore, Student extends the Person class. The keyword extends produces an Inheritance relationship between Student and Person. If I click CTRL + SPACE, we see all of the methods from the Person class. The Student class inherited getDateOfBirth, getName, setDateOfBirth, and setName. Notice, we do not see the private variables name and dateOfBirth. They cannot be seen or accessed by the child class. Inheriting a class does not overrule the access restriction for private properties. Private properties are not visible by outside classes.

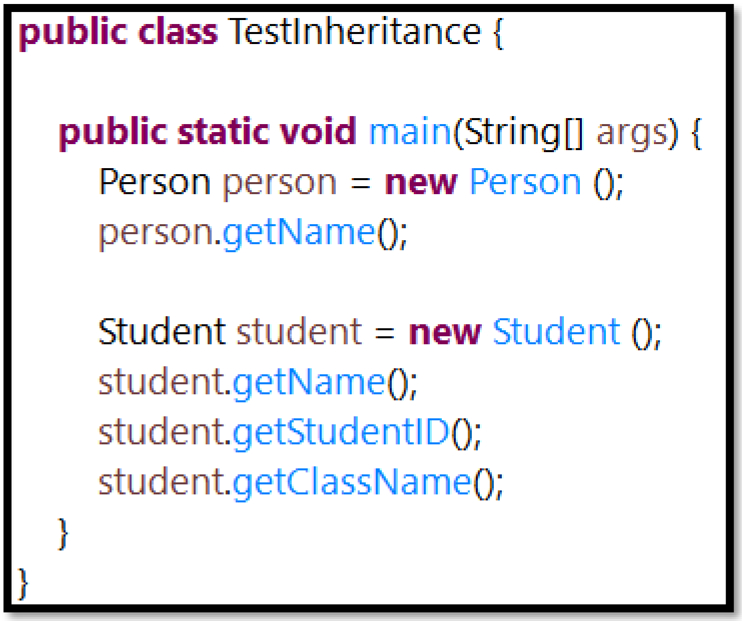
One of the benefits of Inheritance, is the public properties of a parent class can be shared by all child classes. After receiving the public properties, a child class can add its own unique members. Now, we are not concerned about creating a variable and method for name or date of birth in the Student class. This helps the Student class focus on specialized variables and methods that are only used by a Student. For example, we can write private int studentID and private String className. The studentID and className are special to the Student class. Generate the methods again: by right clicking > Source > Generate Getters and Setters > check both boxes and click OK.



```
public class Student extends Person {
   private int studentID;
   private String className;

   public int getStudentID() {
      return studentID;
   }
   public void setStudentID(int studentID) {
      this.studentID = studentID;
   }
   public String getClassName() {
      return className;
   }
   public void setClassName(String className) {
      this.className = className;
   }
}
```

We have our methods. We have our parent class and child class. The next class TestInheritance will access both classes. We create a person by writing Person person = new Person (). A student is created by writing Student student = new Student ().

The person object dot has access to all 4 methods defined in the Person class. Let’s select getName from the Person class. The student object dot has access to all 8 methods which are defined in the Person class and Student class. Let’s also select getName. Do you see how we have access to getName? That’s why we are not concerned about the variable being private because the data can still be accessed through a getter method. How about both methods from the Student class? student.getStudentID and student.getClassName.



```
public class TestInheritance {

  public static void main(String[] args) {
    Person person = new Person ();
    person.getName();

    Student student = new Student ();
    student.getName();
    student.getStudentID();
    student.getClassName();
  }
}
```

This is a perfect example of Inheritance where the child class receives members from a parent class. Next, we will take a look at Polymorphism.

Polymorphism

Introduction

Polymorphism is closely related to Inheritance. You know how Inheritance represents an “is a” relationship. Polymorphism also has that same kind of relationship so UnderGraduate “is a” Student and Student “is a” Person. Therefore, one instance such as UnderGraduate can pick up the behaviors and attributes of more than one class. That’s how Polymorphism maintains the capacity to take on many forms.

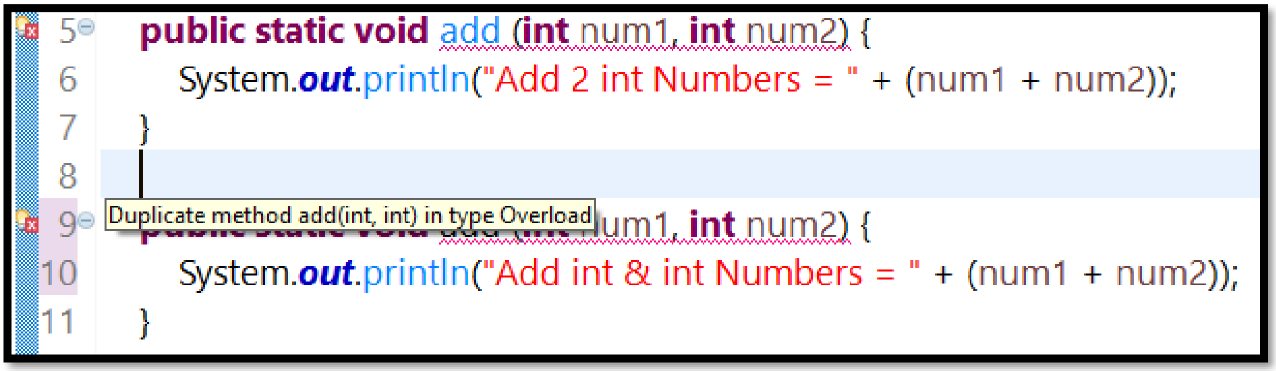
With this diagram, UnderGraduate has 3 forms. UnderGraduate has the form of a Person, UnderGraduate has the form of a Student, and UnderGraduate also has the form of its own type UnderGraduate.

There are 2 ways to implement Polymorphism: Method Overloading and Method Overriding. Method Overloading involves the same class and Method Overriding involves different classes.

In this Polymorphism session, we will investigate Method Overloading, Method Overriding, and both Java Bindings. Method Overloading refers to Static Binding while Method Overriding refers to Dynamic Binding. Let’s start with Method Overloading.

Method Overloading

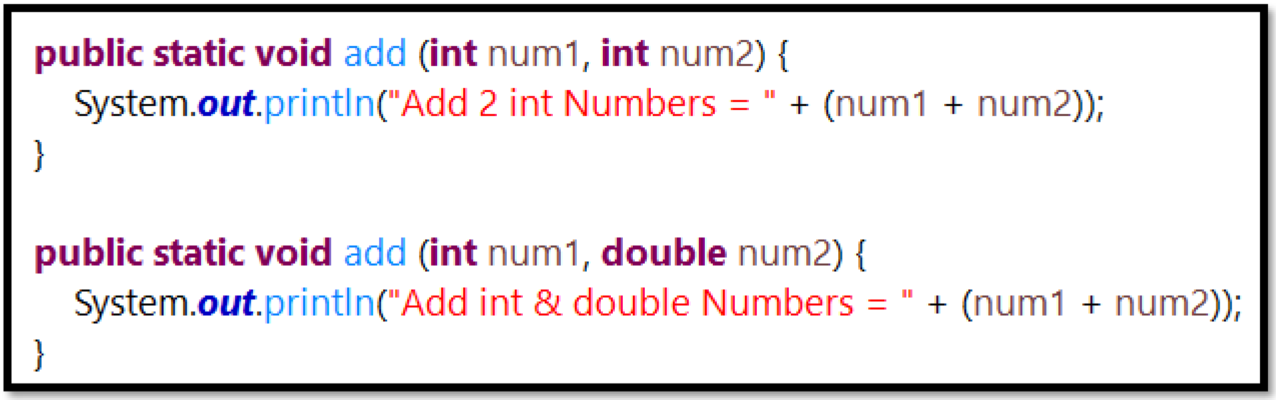
In Java, Method Overloading is when more than 1 method has the same name but different parameters. Go to Eclipse. We see 2 methods named add. However, there’s an error. The error states “Duplicate method add(int, int) in type Overload". That means both methods share the same name and share the same parameter declarations.



```
public static void add (int num1, int num2) {
    System.out.println("Add 2 int Numbers = " + (num1 + num2));
}

public static void add (int num1, int num2) {
    System.out.println("Add int & int Numbers = " + (num1 + num2));
}
```

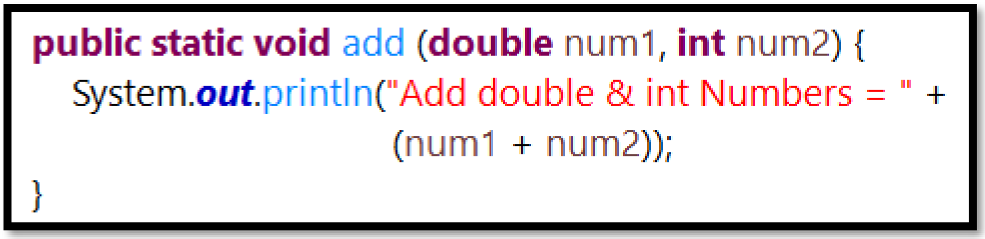
Duplicate method add(int, int) in type Overload

To overload a method, the parameter list must be unique. The type and/or number of parameters must not be the same. Let's start with the type. Change the 2nd type to double, include double in the print statement, save, and now the error goes away. Copy and Paste.



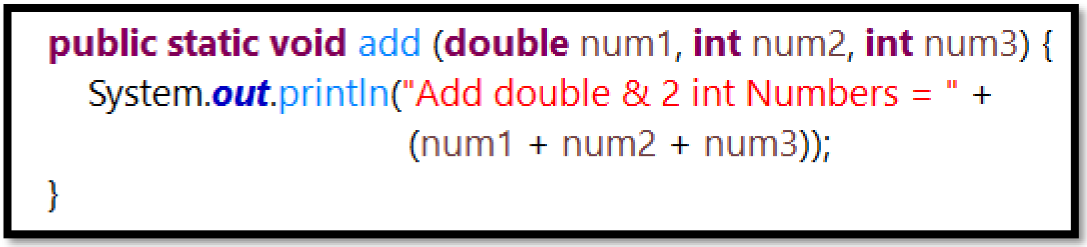
```
public static void add (int num1, int num2) {
    System.out.println("Add 2 int Numbers = " + (num1 + num2));
}

public static void add (int num1, double num2) {
    System.out.println("Add int & double Numbers = " + (num1 + num2));
}
```

We can also swap the types int and double to overload a method. Also swap in the print statements. All 3 methods have a unique order of types. First method int, int. Second method has int, double and Third method has double, int.



```
public static void add (double num1, int num2) {
    System.out.println("Add double & int Numbers = " +
                          (num1 + num2));
}
```

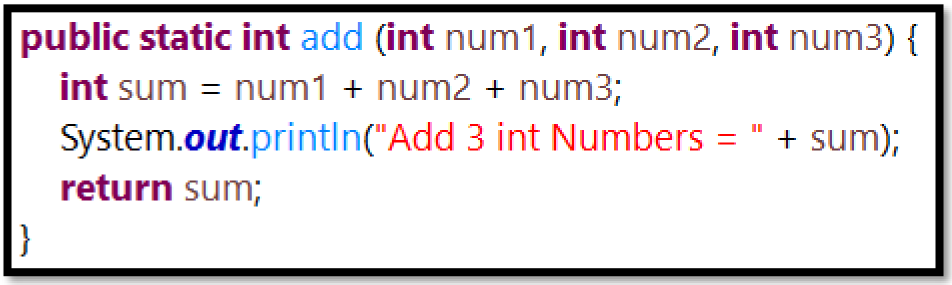
Now, let's look at the number of parameters. Copy and Paste then add int num3. We see every way to overload a method because the type and/or number of parameters are unique. Also update the print statement: 2 int Numbers and include num3 in the parenthesis.



```
public static void add (double num1, int num2, int num3) {
    System.out.println("Add double & 2 int Numbers = " +
                        (num1 + num2 + num3));
}
```

The return type has no impact on overloading a method because Java does not decide which method to execute based on the return type. If I copy and paste

then change the return type to int. Remove the print statement then assign (num1 + num2 + num3) to int sum =. Print statement sysout("Add 3 int Numbers =" + sum) and return statement includes sum. Hover both errors and they show "Duplicate method add(double, int, int) in type Overload". The return types are different but there's still an error. A difference in return types cannot be used as an overloaded method. Change the first type to int and now we have our overloaded method.



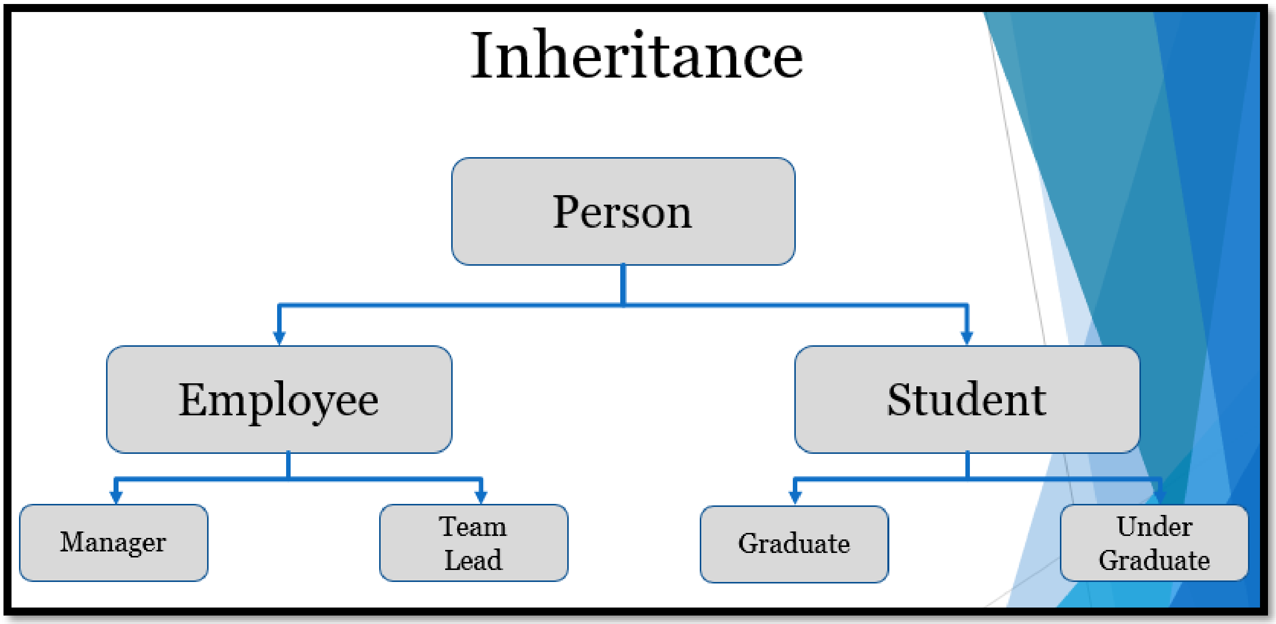
```
public static int add (int num1, int num2, int num3) {
    int sum = num1 + num2 + num3;
    System.out.println("Add 3 int Numbers = " + sum);
    return sum;
}
```

A constructor can be overloaded by using this same principle. Next, we will take a look at Method Overriding.

Method Overriding

Method Overriding allows a superclass to specify a method that will be common to all subclasses. Combining inheritance with method overriding, satisfies the one interface – multiple methods aspect of Polymorphism. A superclass defines the general method that can be used by all of its subclasses. Each subclass is flexible to define its own specialized method while enforcing one consistent interface. Method Overriding only happens when a method in a subclass has the same return type and signature as a method in the superclass.

In our diagram, Student is a child of the Person class. It's also the parent of Graduate and Undergraduate. In real life, Graduate is a student who has a bachelor's degree and pursuing a master's or doctorate degree. An UnderGraduate is a student pursuing a bachelor's or associate degree. Both children classes will automatically inherit the same Student method if the method is public.



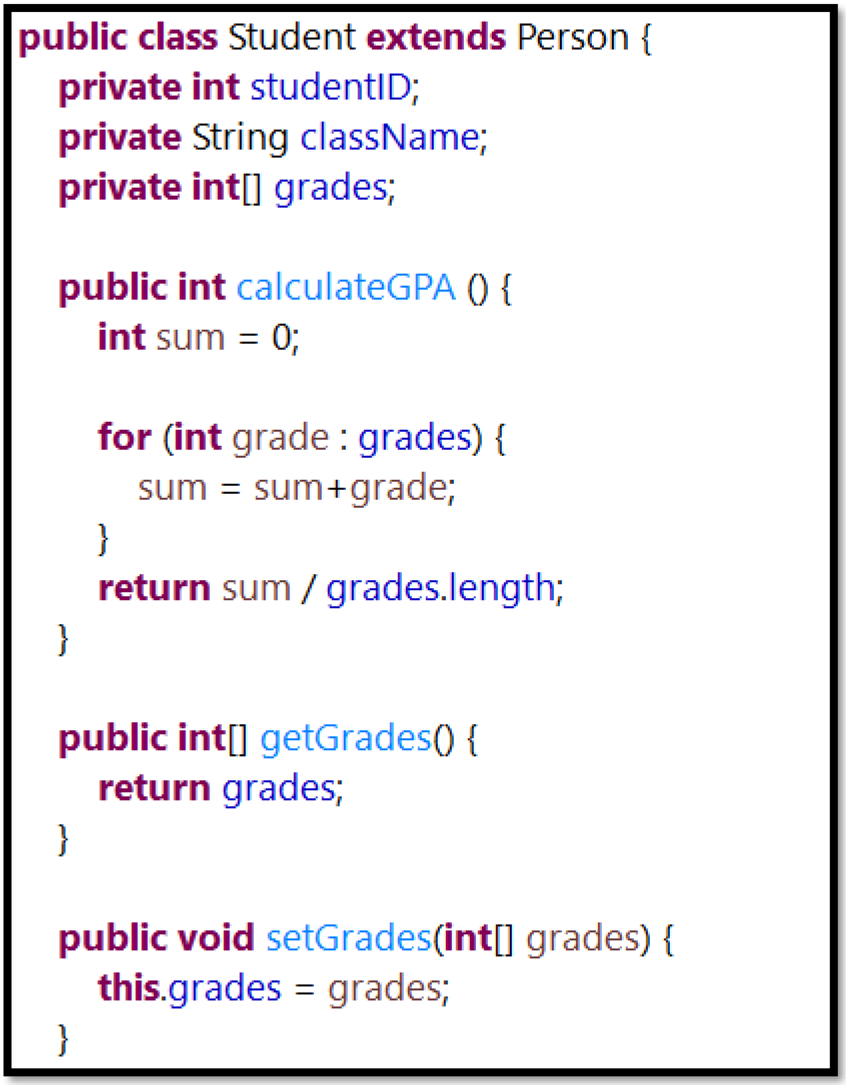
Inheritance

In this session, we already have our Person class and Student class from the previous Inheritance session. The Student class extends Person. Let me minimize these getters and setters. For starters, I will create a method to calculate the grades for each student: public int calculateGPA ()

We have to create an array to store a list for the grades by writing private int[] grades. This method returns 0. Now, let's calculate the GPA. First, we initialize int sum = 0 then a

```
for loop (int grade : grades) {
  sum is assigned to sum + the next grade
}
```

return sum / grades.length. length provides the number of elements in the array. Add getters and setters. Source, Generate Getters and Setters, select grades then OK.



```
public class Student extends Person {
   private int studentID;
   private String className;
   private int[] grades;

   public int calculateGPA () {
      int sum = 0;

      for (int grade : grades) {
         sum = sum+grade;
      }
      return sum / grades.length;
   }

   public int[] getGrades() {
      return grades;
   }

   public void setGrades(int[] grades) {
      this.grades = grades;
   }
```

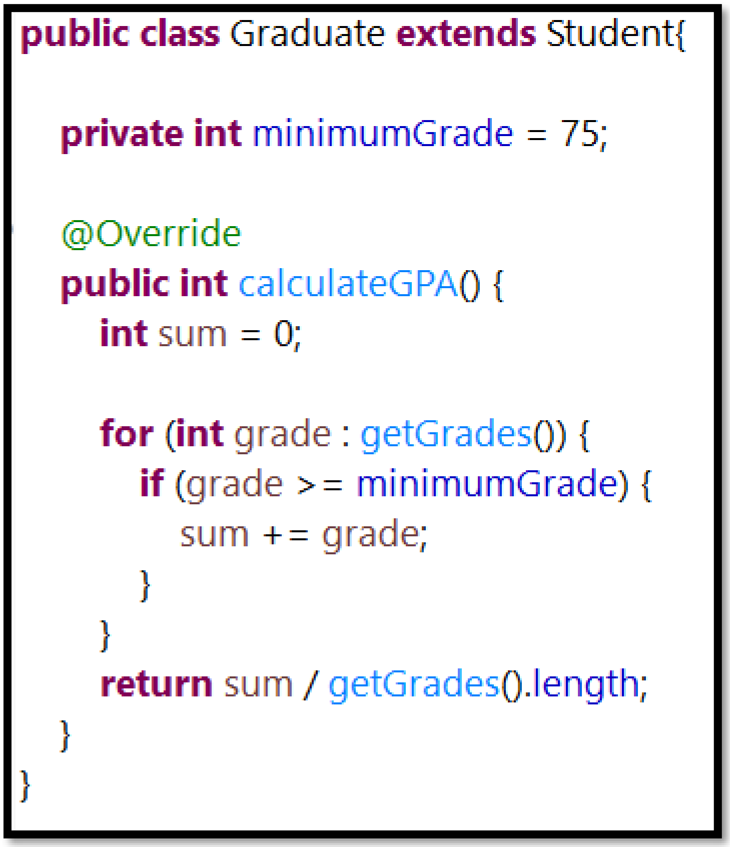
Let’s imagine Graduate students only get credit for grades 75 and above. Start with a private int field called minimumGrade = 75. In addition to minimumGrade, we need a specialized method only for the Graduate class. This method will override the method in the Student class. With Inheritance, Graduate extends Student. CTRL + Space. Do you see how the calculateGPA method and these other methods that override a method in Student? We also see some override methods for the Person class.

Select calculateGPA and erase the comment. It’s not required but its recommended we use this @Override annotation to prevent an error such as providing a different method name. Now, let’s calculate the GPA based on grades 75 and higher.

int sum = 0
for (int grade : call the getGrades()) { method from the Student class
  if (grade >= minimumGrade) {
   sum += grade This is the shortcut syntax for sum = sum + grade

```
 }
}
```

return sum / getGrades().length



```
public class Graduate extends Student{

   private int minimumGrade = 75;

   @Override
   public int calculateGPA() {
      int sum = 0;

      for (int grade : getGrades()) {
         if (grade >= minimumGrade) {
            sum += grade;
         }
      }
      return sum / getGrades().length;
   }
}
```

This calculateGPA method is how we implement a specialized method for a child class to override its parent class. Next, we are going to look at the difference between both Java Bindings.

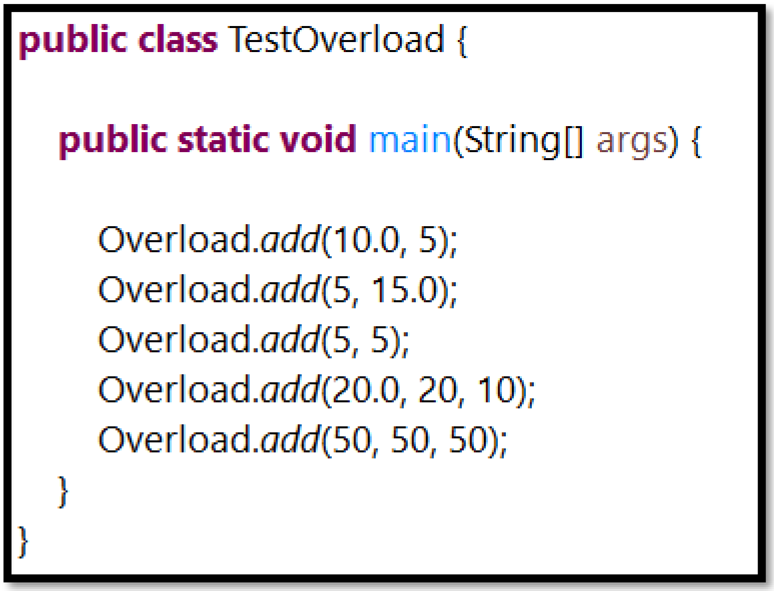
Static vs Dynamic Binding

The Java Bindings are static and dynamic. Static happens at compile time while Dynamic happens at runtime. Method Overloading is an example of Static Binding and Method Overriding is an example of Dynamic Binding.

Let's start with the Static Binding by testing all 5 of these static overloaded methods. Static means they are shared between the whole class. As a result, static methods are bound to their implementations at compile time. The methods can be called directly by its class name. Therefore, it's not important to create an object to access the 5 static methods.

1. We can write the class name Overload. and we see all 5 methods. Select double, int then write 10.0 and 5.
2. Next is Overload. int, double and write 5, 15.0.

3. Third method is Overload. int, int. Let’s write 5, 5.
4. Number 4 has 3 parameters Overload. double, int, int then write 20.0, 20, 10.
5. The last method had an int return type. Overload. Do you see how 4 methods have void as their return type and one method has int? Write 50, 50, 50. Let’s Run. We see a print statement for each overloaded method.



```
public class TestOverload {

   public static void main(String[] args) {

      Overload.add(10.0, 5);
      Overload.add(5, 15.0);
      Overload.add(5, 5);
      Overload.add(20.0, 20, 10);
      Overload.add(50, 50, 50);
   }
}
```

Java restricts static methods from being overridden. Dynamic Binding allows for overridden methods which are multiple implementations that can be called depending on the instance. For example, let’s think about the calculateGPA methods from the Student and Graduate classes. That we did in the previous session for Method Overriding. We can use a separate implementation for Student and for Graduate. However, I am going to implement one Graduate and another one for UnderGraduate.

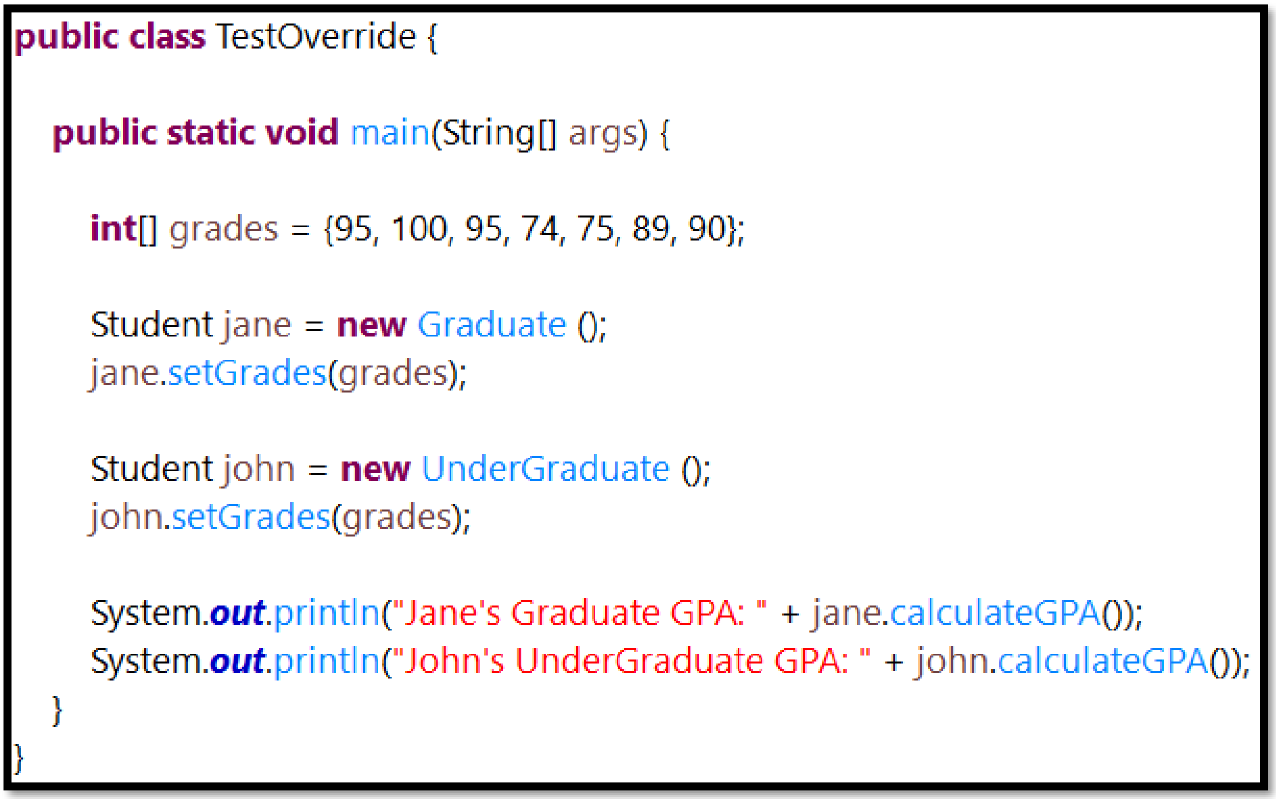
Recall, Graduate has a method to override the Student method. UnderGraduate does not have a method to override Student so it will use the method created in the Student class. Here’s the scenario with 2 instances. Jane is a Graduate and John is an UnderGraduate. We are going to imagine both students have 7 grades and the same grades for each class. They have 7 classes and the same grade for each class. The difference is John gets credit for every class but Jane only gets credit for classes 75 and above.

We have a field for minimumGrade equal to 75 and a calculateGPA method. Let’s create an array for grades: int[] grades = Let’s assign 7 grades. {95, 100, 95, 74, 75, 89, 90}. Bingo, Now our objects.
Student jane = new Graduate()
jane.setGrades(grades)

The next student is John.

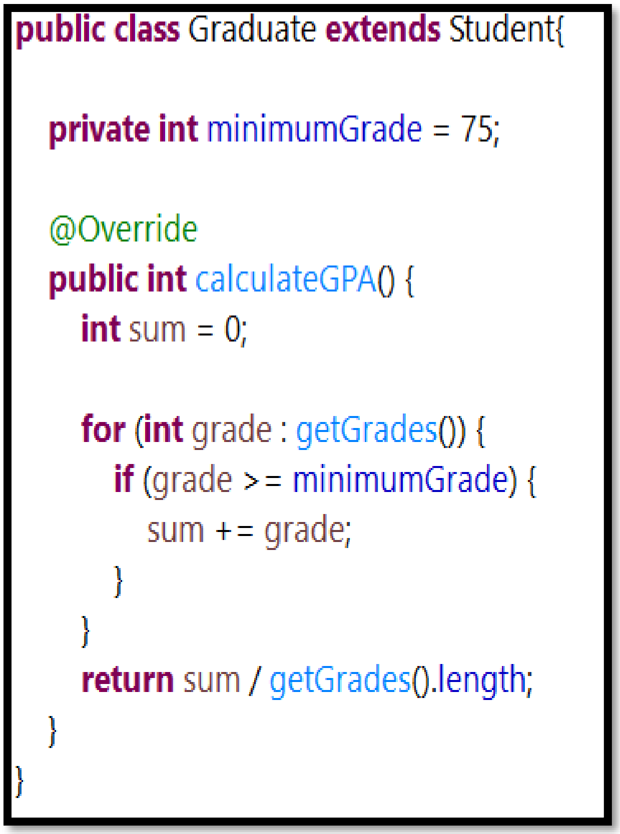
Student john = new UnderGraduate ()
john.setGrades(grades)

Both students have the same grades but Jane gets credit for 6 of the 7 grades. She does not get credit for the 74. Let’s print the grade point average.
sysout(“Janes’ Graduate GPA: ” + jane.calculateGPA())
sysout(“John’s UnderGraduate GPA: ” + john.calculateGPA)

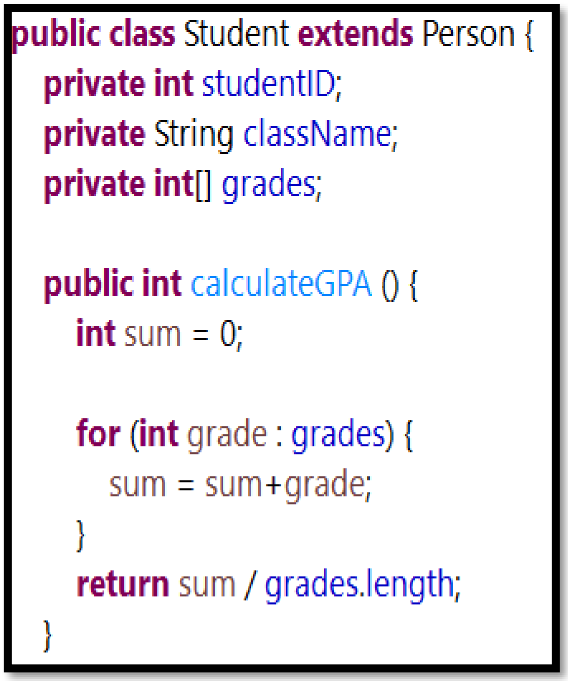


```
public class TestOverride {

  public static void main(String[] args) {

    int[] grades = {95, 100, 95, 74, 75, 89, 90};

    Student jane = new Graduate ();
    jane.setGrades(grades);

    Student john = new UnderGraduate ();
    john.setGrades(grades);

    System.out.println("Jane's Graduate GPA: " + jane.calculateGPA());
    System.out.println("John's UnderGraduate GPA: " + john.calculateGPA());
  }
}
```

Let’s break this down before I run. Here’s what’s going to happen when I execute. Dynamic Binding will look at the Graduate instance type for Jane then call the override calculateGPA method in the Graduate class. For John, the instance type is UnderGraduate but the UnderGraduate class does not have an override method to calculate the GPA. Therefore, it will use the calculateGPA method from the Student class.



```
public class Graduate extends Student{

  private int minimumGrade = 75;

  @Override
  public int calculateGPA() {
    int sum = 0;

    for (int grade : getGrades()) {
      if (grade >= minimumGrade) {
        sum += grade;
      }
    }
    return sum / getGrades().length;
  }
}
```



```
public class Student extends Person {
  private int studentID;
  private String className;
  private int[] grades;

  public int calculateGPA () {
    int sum = 0;

    for (int grade : grades) {
      sum = sum+grade;
    }
    return sum / grades.length;
  }
```

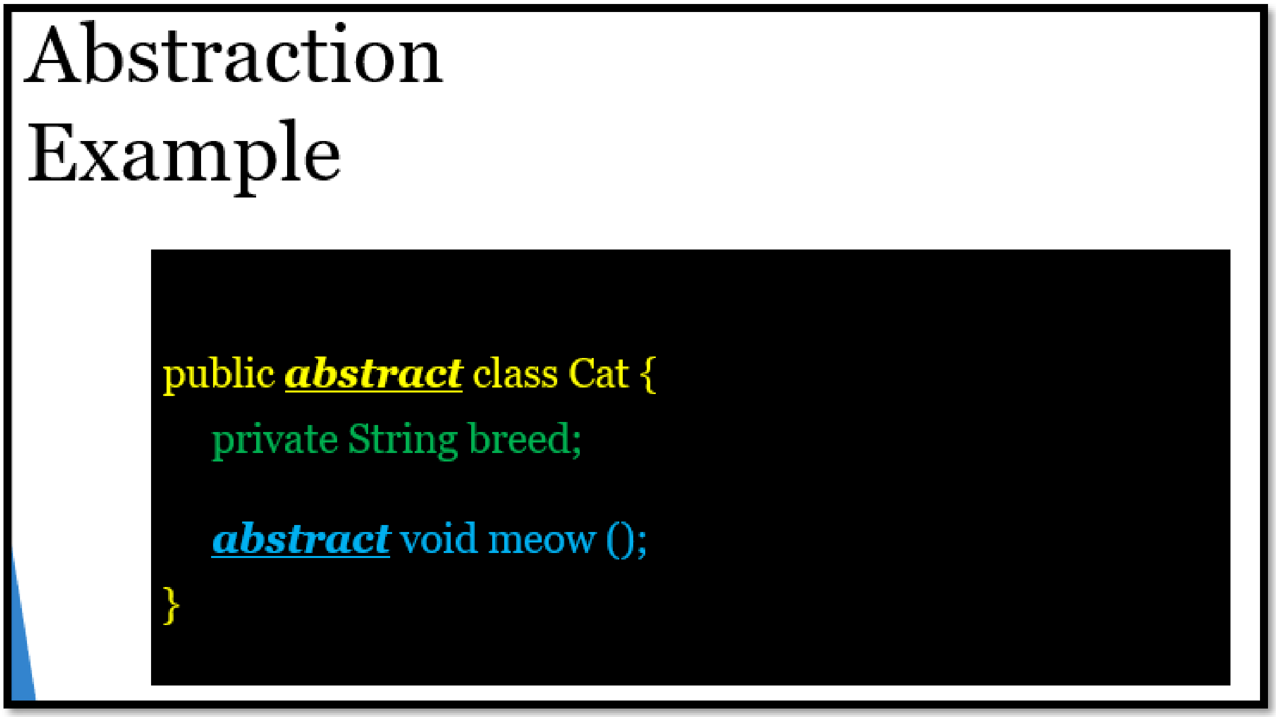
Let's Run. Jane's GPA is 77 while John's GPA is 88. The console shows a different GPA for Jane and John although both students have the same grades. That's what we call Dynamic Binding. We are finished with Polymorphism for Dynamic Binding, Static Binding, Method Overriding, and Method Overloading. In the next session, we will dive into Abstraction.

Abstraction

Introduction

In general, abstract indicates a feature apart from a particular object. When it comes to Object Oriented Programming, abstraction serves as a representative for real things. For example, we have a class named Cat that's abstracted into a type. Next, we declare a field named breed in the Cat class. Breed is a real characteristic that we can abstract from a cat. Last, is the behavior action of a cat. We abstract meow as a real action for a cat.

With this example, abstraction shows the necessary features of an object. Cat is an abstract concept because it displays a general characteristic and a general action. As a result, the type is a Cat with a breed characteristic and meow as an action. Did you notice the keyword abstract?



Abstraction Example

```
public abstract class Cat {
    private String breed;

    abstract void meow ();
}
```

Abstract is a non-access modifier for classes and methods but not for a variable like breed. The purpose of Abstraction is to define a template for a class or define a template for a method. Abstract classes are generic and operate as a superclass. Therefore, it can be extended which allows a subclass to fill in the details.

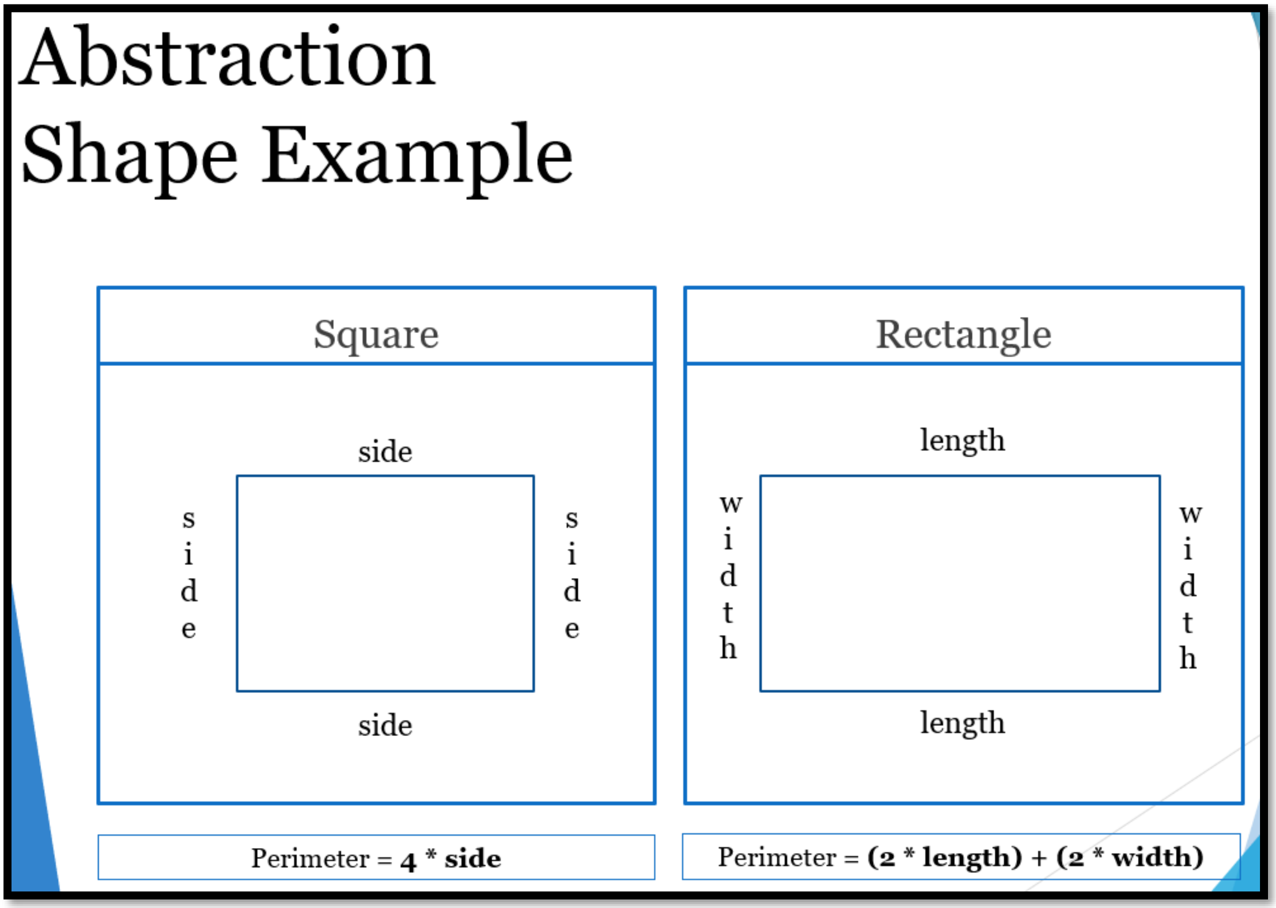
Abstract methods are also generic. They are designed to be overridden in the subclass. That's why we only see the signature of the method: abstract, void, meow, parenthesis, semi-colon, and no body within the curly braces. The details are implemented in the subclass which overrides the abstract method defined in the superclass. Static methods and constructors cannot be abstract.

Java also provide an interface which is similar to an abstract class. One of the differences between an abstract class and interface is: An abstract class can have abstract methods and fully defined methods. The fully defined methods are instance methods which provide a default action.

Traditionally, with an interface, there was not a way to supply a default action. All of the methods were abstract before Java Development Kit JDK 8. Now, we can define a default method to provide a default action. The primary motivation for default methods was to deliver a way to enhance interfaces without breaking existing code. Next, I'm going demo Abstract Classes and Abstract 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.



Abstraction
Shape Example

Square

side

side

side

side

Perimeter = 4 * side

Rectangle

length

width

width

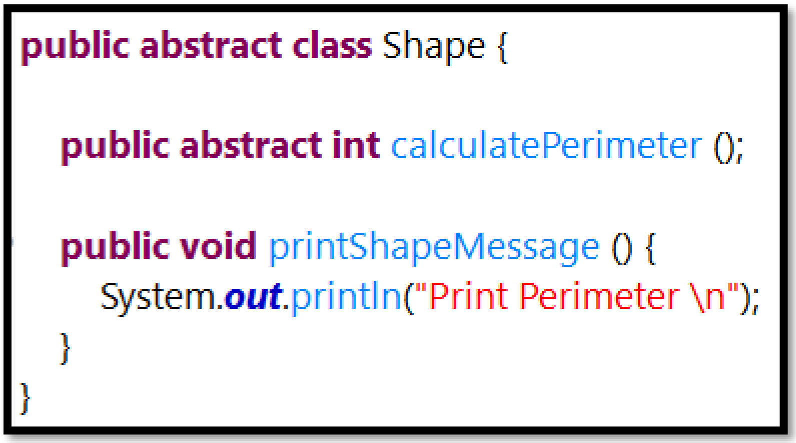
length

Perimeter = (2 * length) + (2 * width)

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.

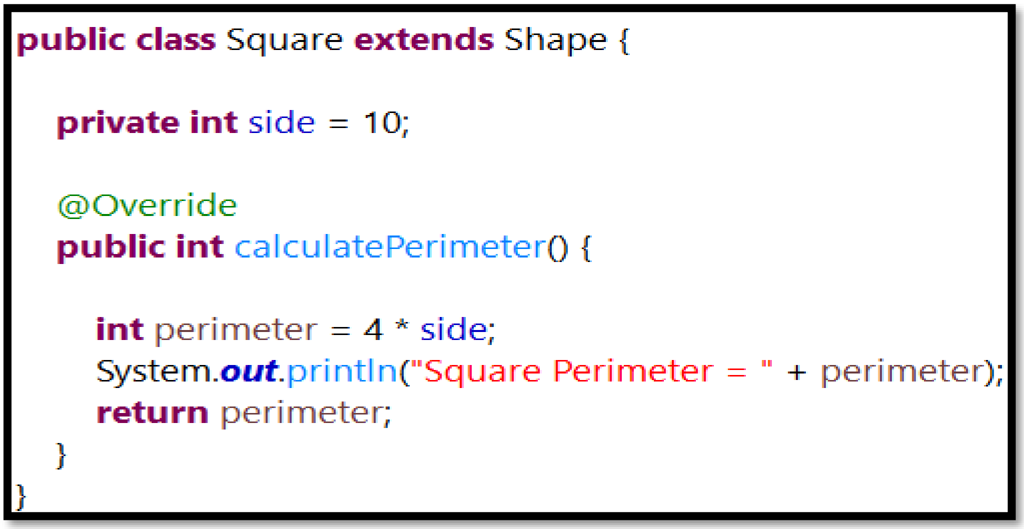
Now there are 2 errors, one at the class level and another one at the method level. Hover the method and we see "The abstract method calculatePerimeter in type Shape can only be defined by an abstract class". If we have one or more abstract methods then we must make the class abstract. That's why we 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”);
}



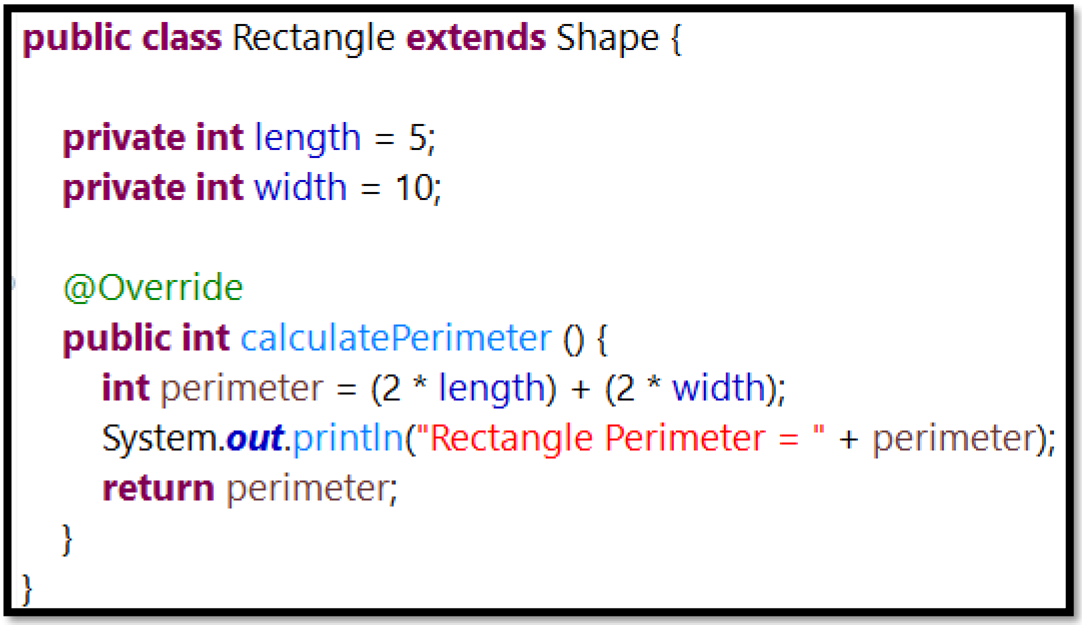
```
public abstract class Shape {

   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;



```
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;
    }
}
```

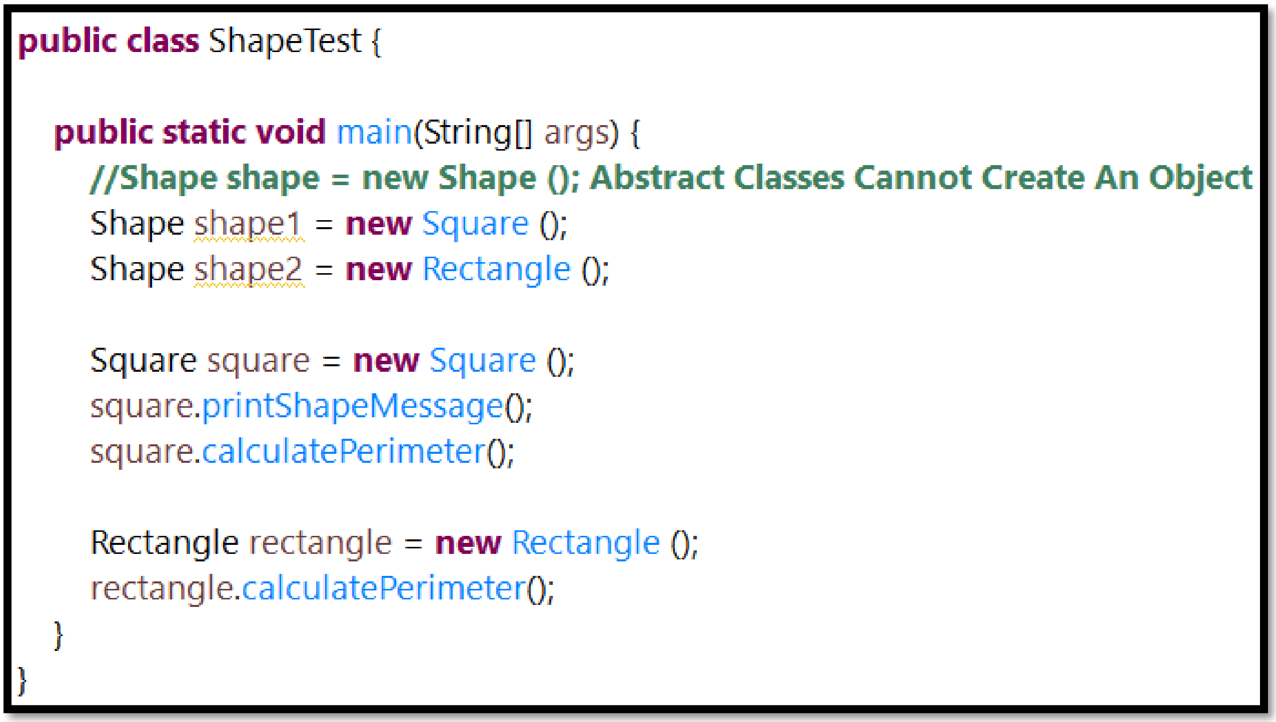
If necessary, you can also create a constructor in an abstract class. That's it for creating an abstract class and abstract methods in a superclass with a subclass implementing the details. Now, let's test out this code in the ShapeTest class. Write main CTRL + SPACE for the shortcut.

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.

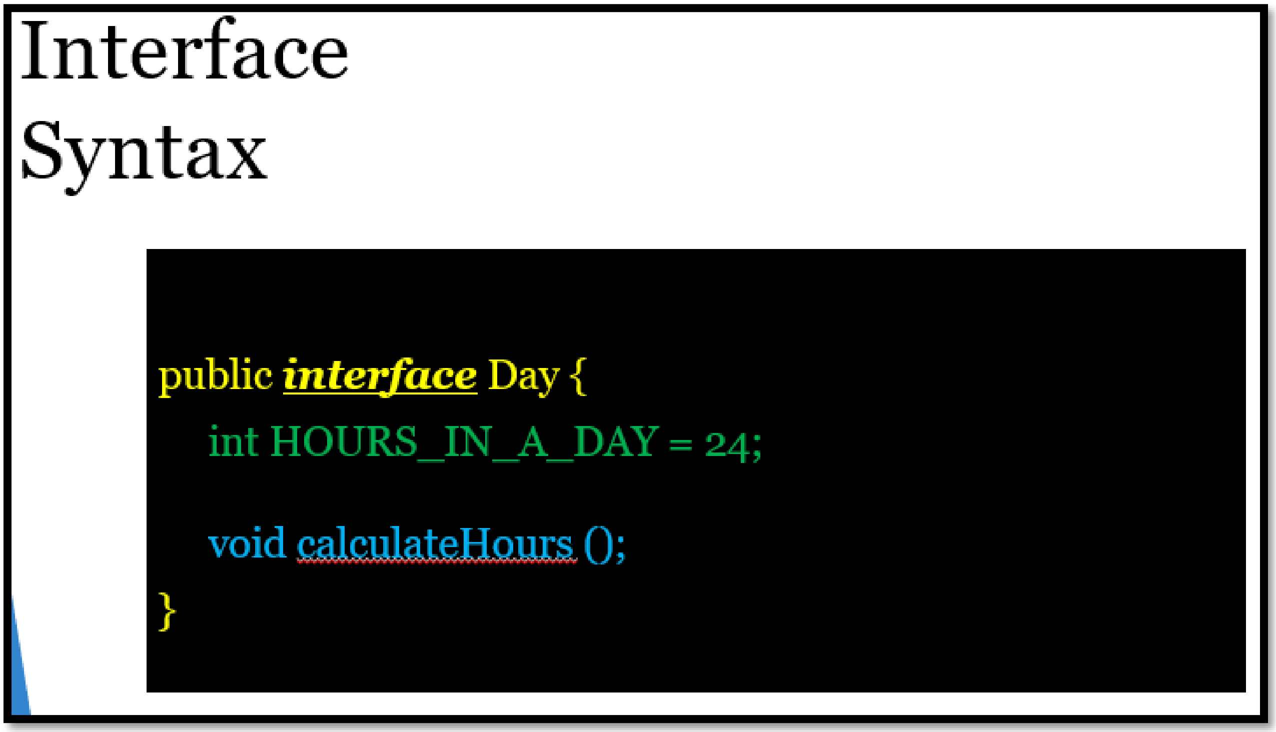
Interfaces

Interface Introduction

An interface is a contract with a class. Contracts are agreements for doing something or not doing something. So, the contract between an interface and class is the interface specifies what a class must do but not how the class must do it. Here's the syntax for an interface. All interfaces have the keyword interface before the interface name. The scope must be public or package. In this example, the scope is public but if we do not define a scope then the default scope is package.

Every field inside in an interface must have a constant value. Therefore, the variable is implicitly public, static, final, and requires a value. In this example, I did not include those 3 keywords: public, static, or final. However, Java will still treat int HOURS_IN_A_DAY = 24 as a constant because interfaces are contracts that cannot hold state. That means it is not allowed to change its behavior. So, we should not add a variable if the value is going to change.

When it comes to methods, they can be abstract, static, or default to an implementation. Notice this method 'void calculateHours' does not include the abstract keyword and it does not include a body. It does not include abstract or a body because the interface assumes all methods are abstract. As a result, the abstract keyword is not required.



Interface Syntax

```
public interface Day {
    int HOURS_IN_A_DAY = 24;

    void calculateHours ();
}
```

Since the interface has a contract with the class, there is a guarantee that each abstract method listed in an interface will be implemented by the class. Static methods and default methods are not required to be implemented by a class because they already have an implementation.

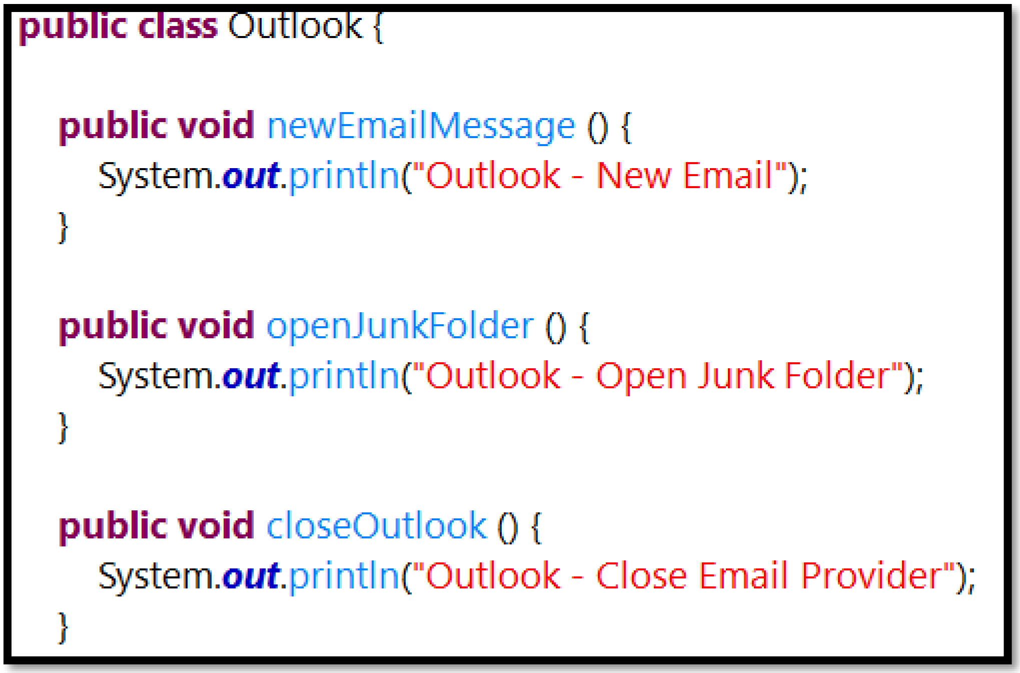
Two of the benefits for using an interface, is to bring consistency for all classes that implement an interface and the support of multiple implementation. My plan is to demo both benefits by starting with consistency for all classes.

How To Implement An Interface

Let's look at 2 email service providers: Outlook and GMail. With Outlook, we see a lot of functions such as File, Home, Send/Receive, Folder, View, and Help. If we wanted to create a new email, we can click New Email. If we wanted to close, we can click X in the top right corner or go to File then click Exit.

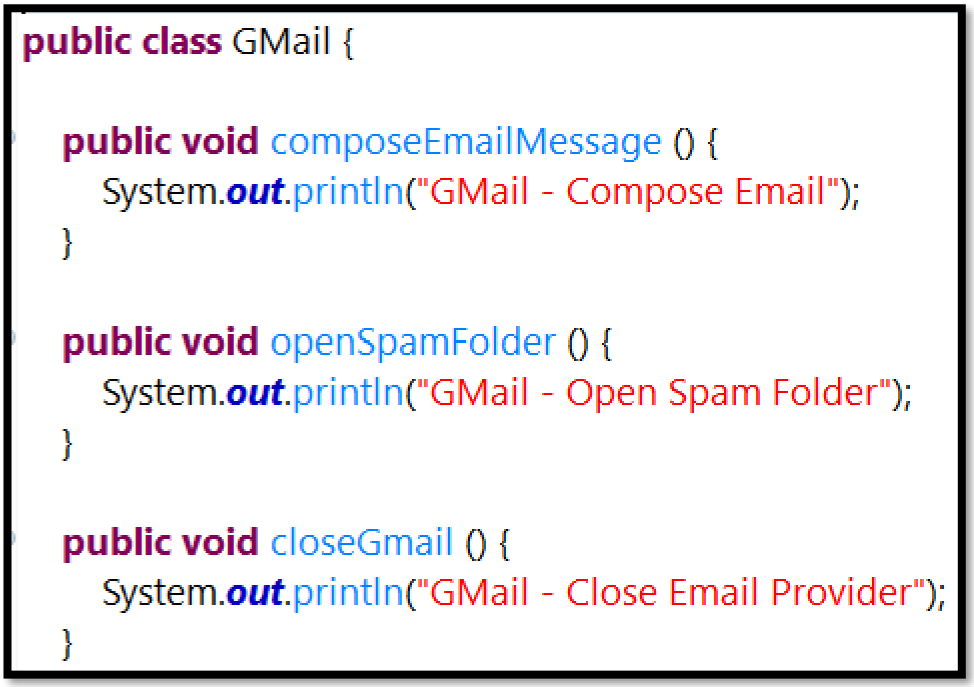
Those same functions are available with GMail and other email service providers. In GMail, to create a new email, we select Compose. It's the same function in Outlook but just a different name. The Spam folder is used to store email that we don't want in our inbox. Outlook has this same function but it's called Junk. Although both providers have functions that operate the same, the name is different and implementation details are different.

For consistency, we can create an interface so Outlook and GMail have the same name with their own unique implementation. Outlook has 3 methods: newEmailMessage, openJunkFolder, and closeOutlook.



```
public class Outlook {

  public void newEmailMessage () {
    System.out.println("Outlook - New Email");
  }

  public void openJunkFolder () {
    System.out.println("Outlook - Open Junk Folder");
  }

  public void closeOutlook () {
    System.out.println("Outlook - Close Email Provider");
  }
```

GMail also has 3 methods but different names: composeEmailMessage, openSpamFolder, and closeGmail.

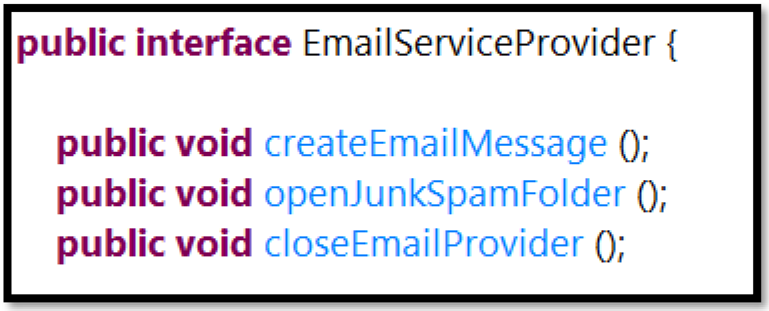


```
public class GMail {

  public void composeEmailMessage () {
    System.out.println("GMail - Compose Email");
  }

  public void openSpamFolder () {
    System.out.println("GMail - Open Spam Folder");
  }

  public void closeGmail () {
    System.out.println("GMail - Close Email Provider");
  }
```

Recall from the introduction, an interface is a contract with a class. It's an agreement that determines what a class must do but not how the class must do it. Therefore, both classes will have the same method name but different details.

To create an interface, we go to New > Interface. The name will be EmailServiceProvider then click Finish. It looks like a class but we see public interface and not public class. We create a method by writing public void then the method name createEmailMessage () add a ; semicolon.

Writing keyword abstract after public is optional. As a convention, we do not write abstract. It's an assumption, that all methods in an interface are abstract although we can write default methods and static methods. For example, if I remove the semicolon then add a body which start and stop with the curly braces. An error states "Abstract methods do not specify a body" but show 3 quick fixes which include changing the method to default or static. Remove the braces then add back the semicolon. Abstract methods define the signature of a method but no implementation for the method. Some more methods that we can add to this interface are public void openJunkSpamFolder (); and public void closeEmailProvider ();.

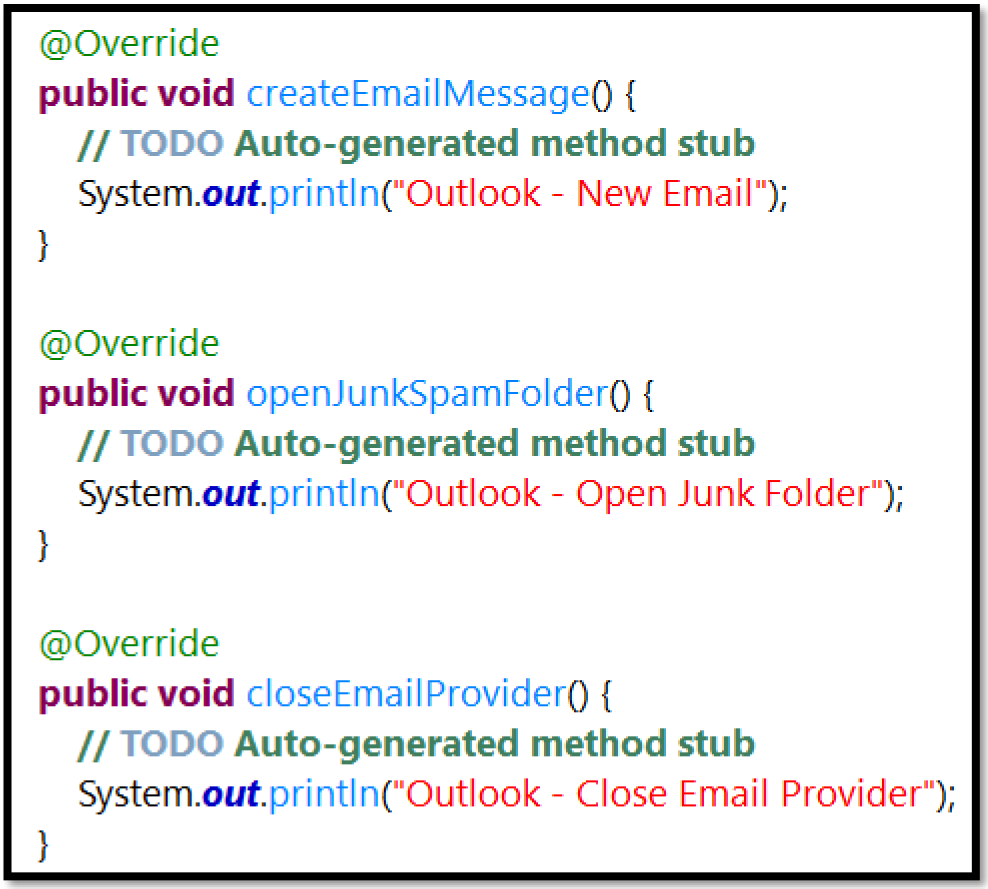


```
public interface EmailServiceProvider {

   public void createEmailMessage ();
   public void openJunkSpamFolder ();
   public void closeEmailProvider ();
```

So, this interface says; all implementation classes should be able to create an email message, open a junk spam folder, and close the email provider.

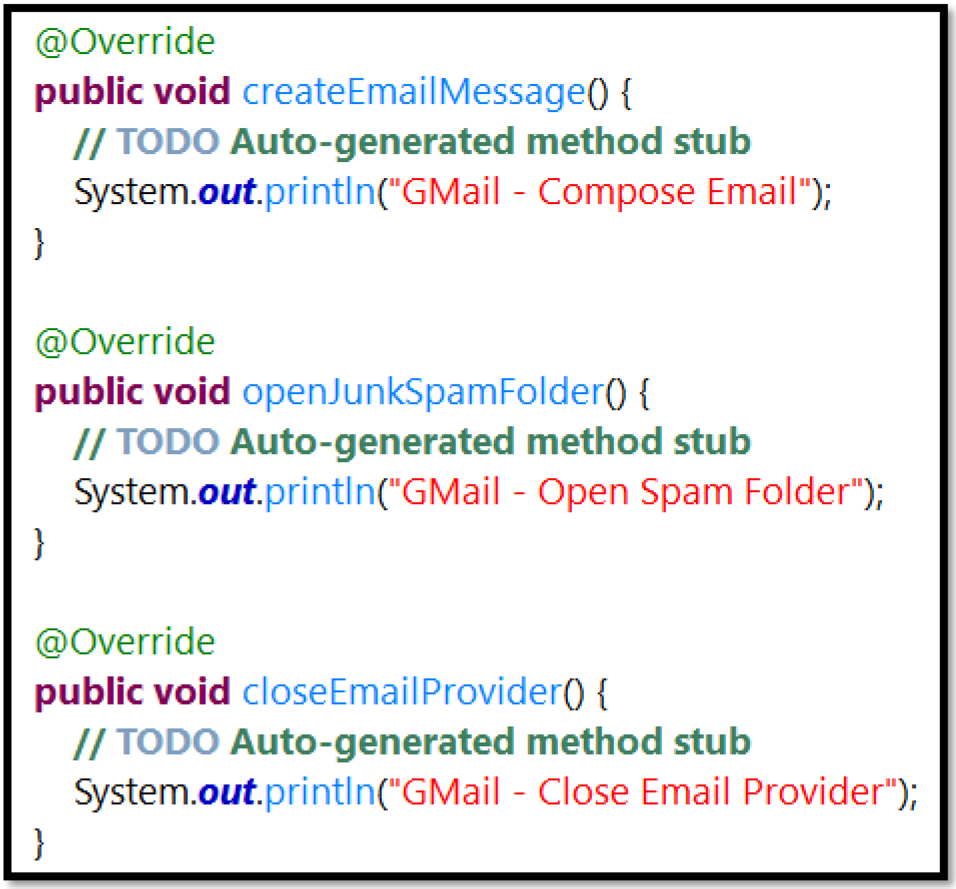
Now, let's implement the interface. Go to Outlook and write implements EmailServiceProvider. Instantly, we see an error "The type Outlook must implement the inherited abstract method EmailServiceProvider.openJunkSpamFolder()". All 3 methods are recognized as an abstract method. The 2 quick fixes are Add unimplemented methods and Make type 'Outlook' abstract. Let's add the unimplemented methods. We see each method from the interface EmailServiceProvider with an Override annotation.

This is where the Outlook class can implement its unique details for each method. I'm going to copy and paste the print statements. Outlook – New Email, Outlook – Open Junk Folder, and Outlook – Close Email.



```
@Override
public void createEmailMessage() {
    // TODO Auto-generated method stub
    System.out.println("Outlook - New Email");
}

@Override
public void openJunkSpamFolder() {
    // TODO Auto-generated method stub
    System.out.println("Outlook - Open Junk Folder");
}

@Override
public void closeEmailProvider() {
    // TODO Auto-generated method stub
    System.out.println("Outlook - Close Email Provider");
}
```

The same for Gmail, it implements EmailServiceProvider and let's Make the type abstract. The keyword is added before class. We should not make the class abstract because it can lead to an error when creating an object. Objects cannot instantiate an abstract type. Remove abstract, save, then select Add unimplemented methods. Also copy and paste the print statements. GMail – Compose Email, GMail – Open Spam Folder, and GMail – Close Email Provider.



```
@Override
public void createEmailMessage() {
    // TODO Auto-generated method stub
    System.out.println("GMail - Compose Email");
}

@Override
public void openJunkSpamFolder() {
    // TODO Auto-generated method stub
    System.out.println("GMail - Open Spam Folder");
}

@Override
public void closeEmailProvider() {
    // TODO Auto-generated method stub
    System.out.println("GMail - Close Email Provider");
}
```

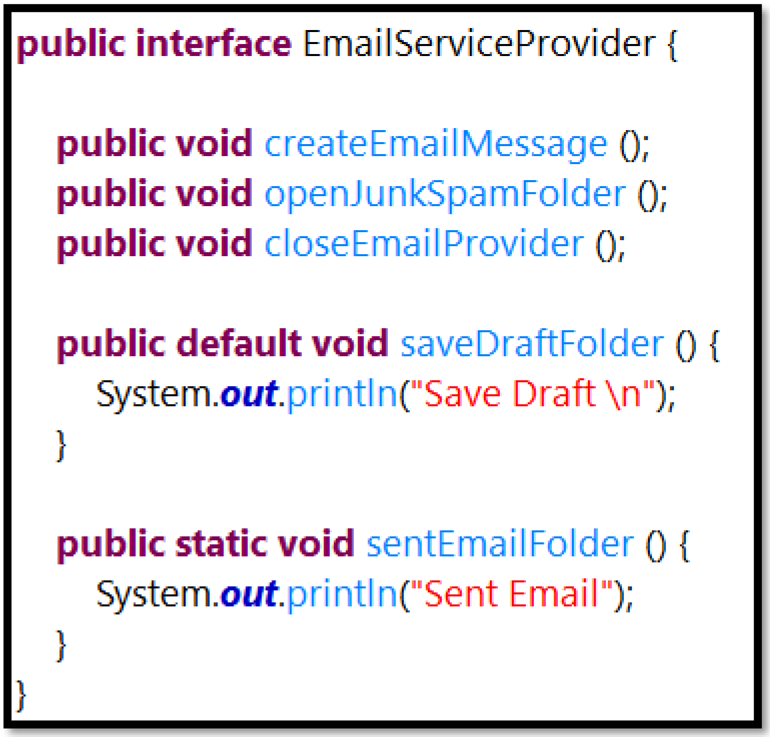
The contract agreement between the interface and each class has been fulfilled. Look what happens when the interface breaks the contract agreement by adding another abstract method.

public void saveDraftFolder(); then click Save. We see an error in the Outlook class and GMail class. The error exists because both classes was not part of the original agreement with saveDraftFolder.

To solve this error, we can do 1 of 2 things. First, we can go to each implementation class and implement the abstract method but what if our program had more than 2 classes? What if I wanted to add another method later on? I prefer option 2 and extend the interface. Java allows us to extend the interface so we don't have the fear of breaking an implementation class by creating a default method. The key is to add default and include a body. There will be an error if we only add default. The error states "This method requires a body instead of a semicolon". Add a print statement for the body by writing sysout("Save Draft \n") then Save. Both errors are gone but if we wanted to, we could override the default method.

However, when it comes to a static method, it provides a protection for an interface. It provides protection by not allowing an implementation class to override the static method. You can watch the Java Bindings video where I speak about how Java restricts static methods from being overridden. We

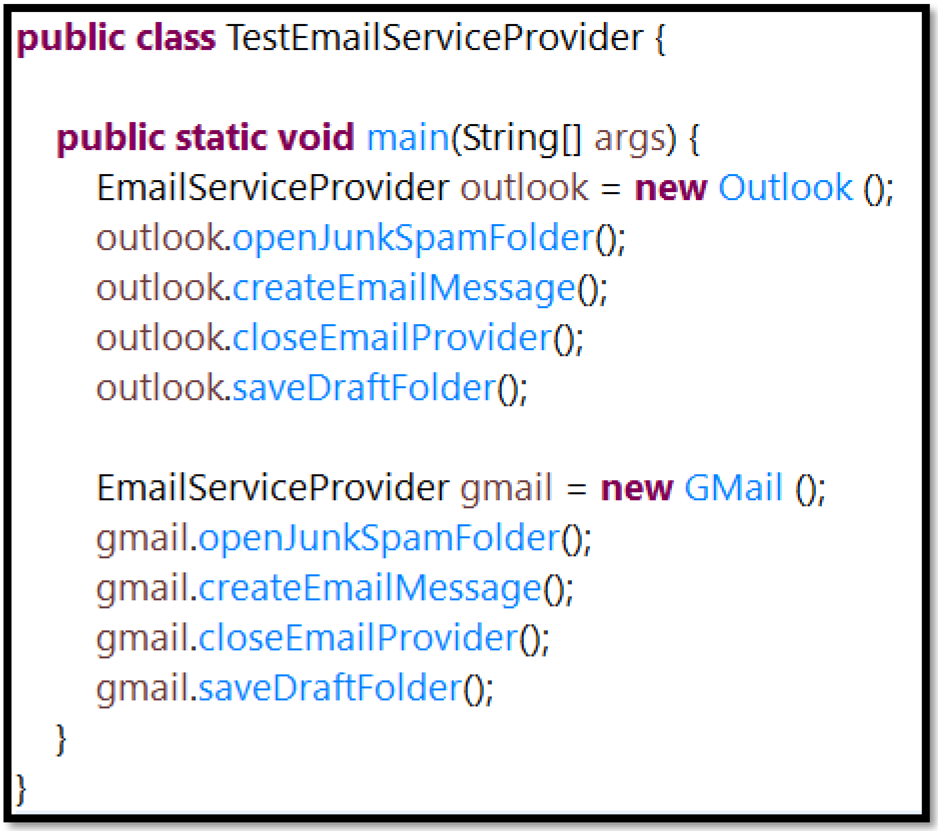
can write public static void sentEmailFolder () {
sysout("Sent Email") }.



```
public interface EmailServiceProvider {

   public void createEmailMessage ();
   public void openJunkSpamFolder ();
   public void closeEmailProvider ();

   public default void saveDraftFolder () {
      System.out.println("Save Draft \n");
   }

   public static void sentEmailFolder () {
      System.out.println("Sent Email");
   }
}
```

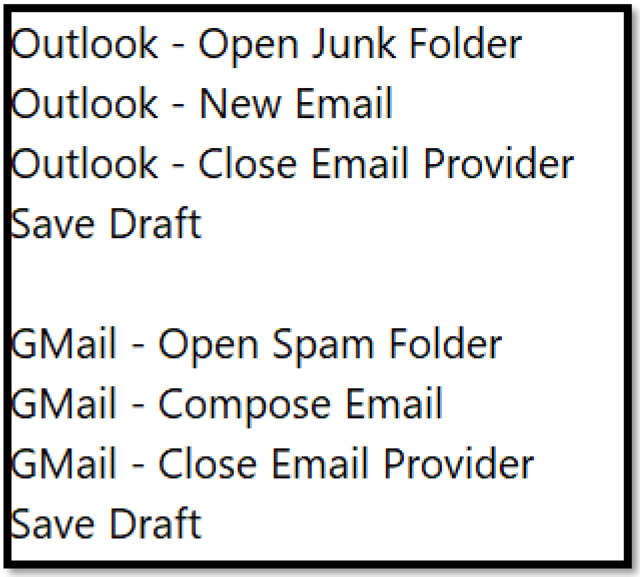
Let's test the interface and both implementation classes. Go to New > Class > Main > TestEmailServiceProvider then Finish. EmailServiceProvider object outlook = new Outlook (); outlook. (dot) and we see 4 methods. All 3 abstract methods and the default method (closeEmailProvider, createEmailMessage, openJunkSpamFolder and saveDraftFolder). Do you see how the static method is not here?. Select openJunkSpamFolder / outlook.createEmailMessage / outlook.closeEmailProvider / outlook.saveDraftFolder.

We can write the same for GMail. EmailServiceProvider gmail = new GMail (); / gmail. openJunkSpamFolder / gmail.createEmailMessage / gmail.closeEmailProvider / gmail.saveDraftFolder.



```
public class TestEmailServiceProvider {

  public static void main(String[] args) {
    EmailServiceProvider outlook = new Outlook ();
    outlook.openJunkSpamFolder();
    outlook.createEmailMessage();
    outlook.closeEmailProvider();
    outlook.saveDraftFolder();

    EmailServiceProvider gmail = new GMail ();
    gmail.openJunkSpamFolder();
    gmail.createEmailMessage();
    gmail.closeEmailProvider();
    gmail.saveDraftFolder();
  }
}
```

Let’s run. We see 8 print statements. 4 for Outlook and 4 for GMail.

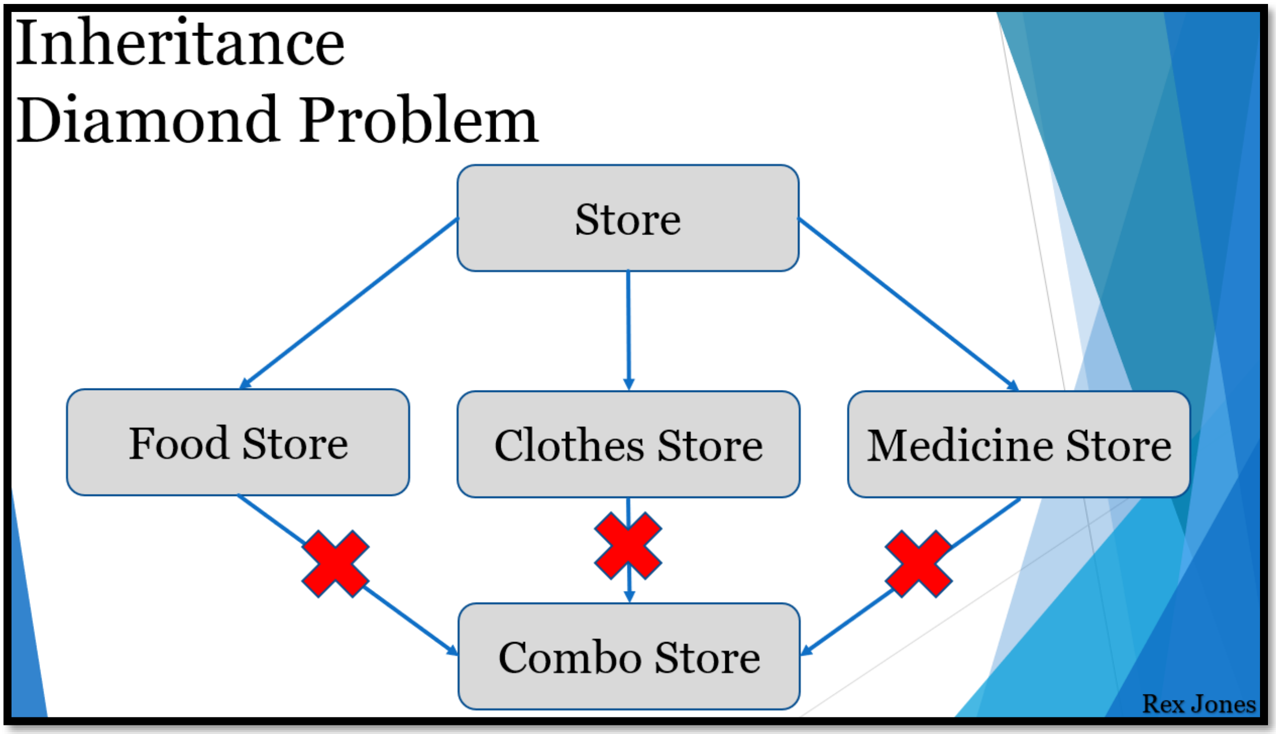


```
Outlook - Open Junk Folder
Outlook - New Email
Outlook - Close Email Provider
Save Draft

GMail - Open Spam Folder
GMail - Compose Email
GMail - Close Email Provider
Save Draft
```

Next, we are going to look at how Interfaces support multiple implementation.

Multiple Interface Implementation and Extension

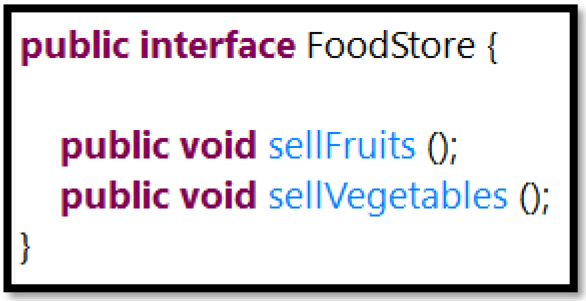
Multiple implementation of interfaces is a way for a class to implement more than 1 interface. That's important because inheritance cannot extend more than 1 class but an interface can extend more than 1 interface. In this diagram, we see 1 Super Class and 3 Sub Classes. Each Sub Class inherits the Super Class. Let's look at this diagram using a Store example. Store is the parent class while Food Store, Clothes Store, and Medicine Store are the child classes. With inheritance, there is no way to combine all stores into 1 Combo Store. It is impossible because we are not allowed to extend more than 1 class. This is called a diamond problem.



Inheritance
Diamond Problem

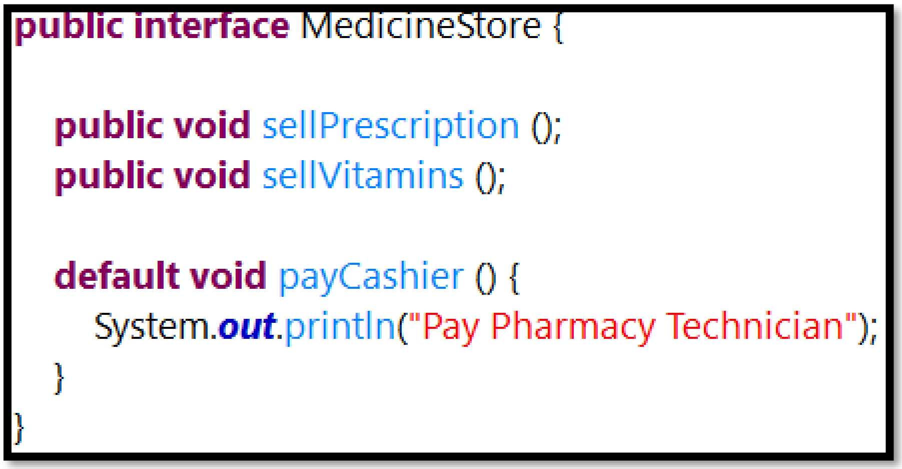
Do you see the diamond? Combo Store will not inherit Food, Clothes, or Medicine although it's possible in real life. However, we can combine all stores when 1 class which is Combo Store implements more than 1 store. There is no limit to the number of interfaces a class can implement.

In Eclipse, we see the FoodStore has 2 abstract methods: sellFruits and sellVegetables.



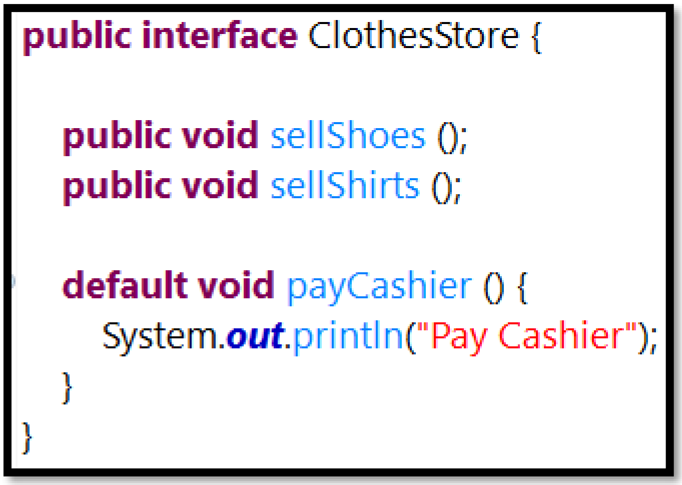
```
public interface FoodStore {

   public void sellFruits ();
   public void sellVegetables ();
}
```

The Medicine Store also has 2 abstract methods and 1 default method: sellPresecription and sellVitamins are the abstract methods, ‘payCashier’ is the default method with details to pay a Pharmacy Technician.



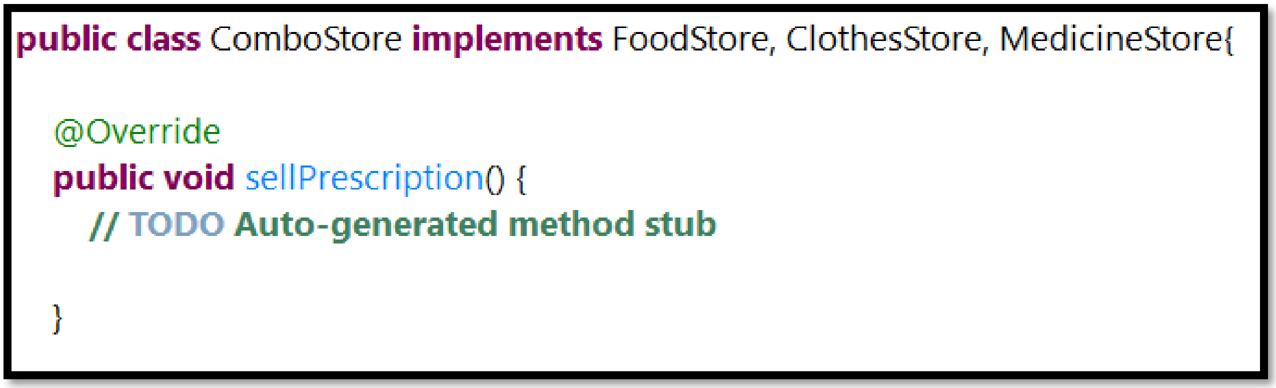
```
public interface MedicineStore {

   public void sellPrescription ();
   public void sellVitamins ();

   default void payCashier () {
      System.out.println("Pay Pharmacy Technician");
   }
}
```

Clothes Store is similar to Medicine Store. It has 2 abstract methods: sellShoes and sellShirts and the same default method named ‘payCashier’ but different implementation details.



```
public interface ClothesStore {

   public void sellShoes ();
   public void sellShirts ();

   default void payCashier () {
      System.out.println("Pay Cashier");
   }
}
```

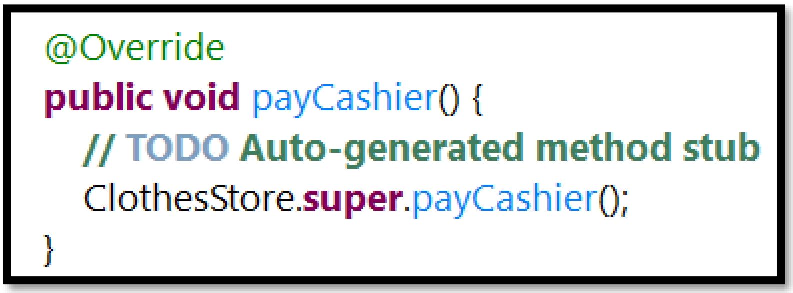
To combine all 3 interface stores, we go to the Combo Store then it implements FoodStore, ClothesStore, and MedicineStore. Just like when implementing 1 interface, we must add all unimplemented methods. There should be 6 overridden methods: sellPrescription, sellVitamins, sellShoes, sellShirts, sellFruits, and sellVegetables.



```
public class ComboStore implements FoodStore, ClothesStore, MedicineStore{

   @Override
   public void sellPrescription() {
      // TODO Auto-generated method stub

   }
```

Save and there's 1 more error "Duplicate default methods named payCashier with the parameters () and () are inherited from the types MedicineStore and ClothesStore".

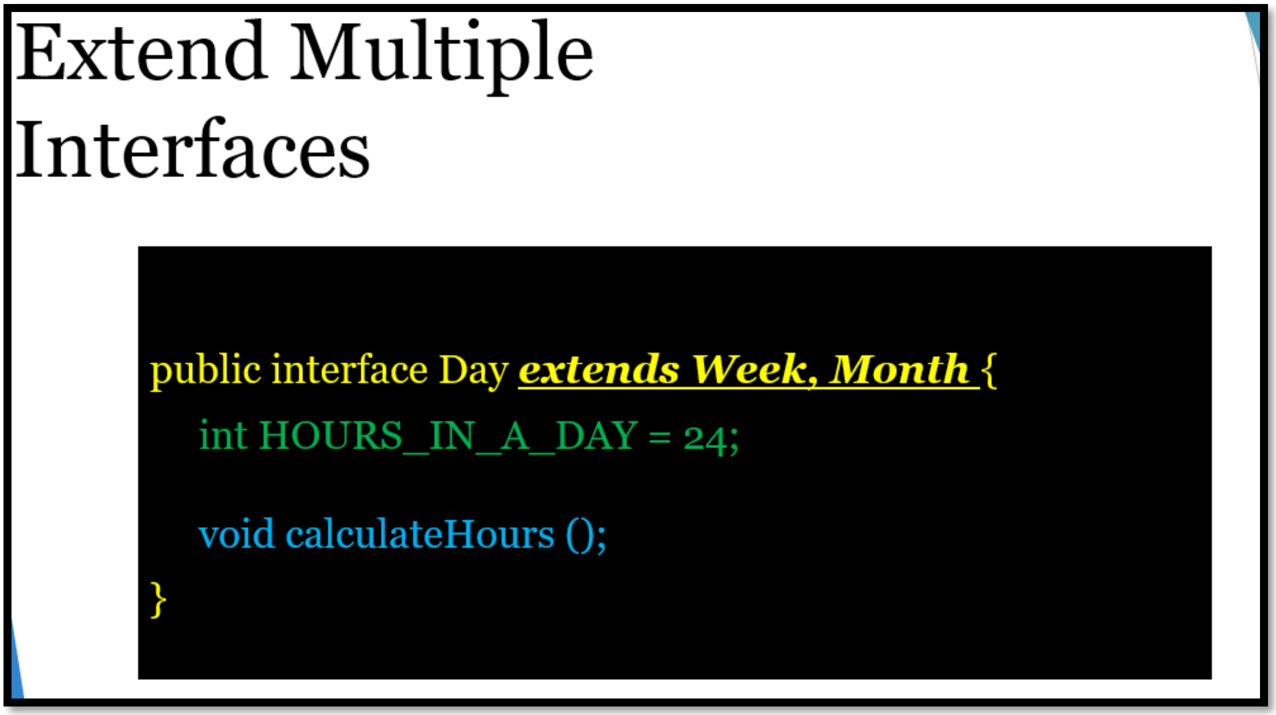
For a quick fix, we can override the default method or change the parameters. This is also a diamond problem because both interfaces have the same default method name payCashier and the compiler cannot decide which super method to use. Override payCashier from ClothesStore. Go to the method and we see Override – ClothesStore.super.payCashier().



```
@Override
public void payCashier() {
   // TODO Auto-generated method stub
   ClothesStore.super.payCashier();
}
```

Recall that a class cannot extend more than 1 class but an interface can extend more than 1 interface. Let's go to the FoodStore interface. It extends another interface called EmailServiceProvider which is another interface from the previous session How To Implement An Interface.

Save and ComboStore will have a different error because the abstract methods from EmailServiceProvider must be implemented. Add unimplemented methods then go to the bottom and we see each method: createEmailMessage, openJunkSpamFolder, closeEmailProvider. In this example, multiple interfaces are extended after adding a comma and another interface name. The interface is Day then it extends 2 interfaces: Week and Month.



Extend Multiple Interfaces

```
public interface Day extends Week, Month {
    int HOURS_IN_A_DAY = 24;

    void calculateHours ();
}
```

That completes the Java Object-Oriented Programming Series. We covered 4 traits of the OOP's concept: Encapsulation, Inheritance, Polymorphism, and Abstraction. I hope you learned something from this OOP series. If you did, Subscribe, Connect, and Follow me because I have so much more content to share with our Testing Community.