Polymorphism using method overloading

```
class Pattern {
  // method without parameter
  public void display() {
    for (int i = 0; i < 10; i++) {
      System.out.print("*");
    }
  }
  // method with single parameter
  public void display(char symbol) {
    for (int i = 0; i < 10; i++) {
      System.out.print(symbol);
    }
  }
}
class Main {
  public static void main(String[] args) {
    Pattern d1 = new Pattern();
    d1.display();
    System.out.println("\n");
      d1.display('#');
  }
}
```

Note: The method that is called is determined by the compiler. Hence, it is also known as compile-time polymorphism.

Java Encapsulation

Encapsulation is one of the key features of object-oriented programming. Encapsulation refers to the bundling of fields and methods inside a single class.

It prevents outer classes from accessing and changing fields and methods of a class. This also helps to achieve data hiding.

```
class Area {
  // fields to calculate area
  int length;
  int breadth;
  // constructor to initialize values
  Area(int length, int breadth) {
    this.length = length;
    this.breadth = breadth;
  }
  // method to calculate area
  public void getArea() {
    int area = length * breadth;
    System.out.println("Area: " + area);
  }
}
class Main {
  public static void main(String[] args) {
    // create object of Area
    // pass value of length and breadth
    Area rectangle = new Area(5, 6);
    rectangle.getArea();
  }
}
```

Note: People often consider encapsulation as data hiding, but that's not entirely true.

Encapsulation refers to the bundling of related fields and methods together. This can be used to achieve data hiding. Encapsulation in itself is not data hiding.

Data Hiding

Data hiding is a way of restricting the access of our data members by hiding the implementation details.

```
class Person {
  // private field
  private int age;
  // getter method
  public int getAge() {
    return age;
  }
  // setter method
  public void setAge(int age) {
    this.age = age;
  }
}
class Main {
  public static void main(String[] args) {
    // create an object of Person
    Person p1 = new Person();
    // change age using setter
    p1.setAge(24);
    // access age using getter
    System.out.println("My age is " + p1.getAge());
  }
}
```

Java Nested and Inner Class

In Java, you can define a class within another class. Such class is known as nested class. For example,

```
class OuterClass {
   // ...
   class NestedClass {
      // ...
   }
}
```

There are two types of nested classes you can create in Java.

1. Non-static nested class (inner class)
2. Static nested class

Non-Static Nested Class (Inner Class)

A non-static nested class is a class within another class. It has access to members of the enclosing class (outer class). It is commonly known as inner class.

Since the inner class exists within the outer class, you must instantiate the outer class first, in order to instantiate the inner class.

```
class CPU {
   double price;
   // nested class
   class Processor{
      // members of nested class
      double cores;
      String manufacturer;
      double getCache(){
         return 4.3;
      }
   }
   // nested protected class
   protected class RAM{
      // members of protected nested class
      double memory;
      String manufacturer;
      double getClockSpeed(){
         return 5.5;
      }
   }
}
public class Main {
   public static void main(String[] args) {

      // create object of Outer class CPU
      CPU cpu = new CPU();

     // create an object of inner class Processor using outer class
      CPU.Processor processor = cpu.new Processor();

      // create an object of inner class RAM using outer class CPU
      CPU.RAM ram = cpu.new RAM();
      System.out.println("Processor Cache = " + processor.getCache());
      System.out.println("Ram Clock speed = " + ram.getClockSpeed());
   }
}
```

Note: We use the dot (.) operator to create an instance of the inner class using the outer class.

Java Nested Static Class

We use the keyword static to make our nested class static.

Note: In Java, only nested classes are allowed to be static.

Like regular classes, static nested classes can include both static and non-static fields and methods. For example,

```
Class Animal {
  static class Mammal {
    // static and non-static members of Mammal
  }
  // members of Animal
}
```

Static nested classes are associated with the outer class.
To access the static nested class, we don't need objects of the outer class.

```
class Animal {
// inner class
  class Reptile {
    public void displayInfo() {
     System.out.println("I am a reptile.");
    }
  }
// static class
  static class Mammal {
    public void displayInfo() {
     System.out.println("I am a mammal.");
    }
  }
}
class Main {
  public static void main(String[] args) {
    // object creation of the outer class
    Animal animal = new Animal();
    // object creation of the non-static class
    Animal.Reptile reptile = animal.new Reptile();
    reptile.displayInfo();
   // object creation of the static nested class
    Animal.Mammal mammal = new Animal.Mammal();
    mammal.displayInfo();
  }
}
```

Static Top-level Class

As mentioned only nested classes can be static. We cannot have static top-level classes.

```
static class Animal {
 public static void displayInfo() {
   System.out.println("I am an animal");
 }
}

class Main {
 public static void main(String[] args) {
   Animal.displayInfo();
 }
}
```

Java Anonymous Class

In Java, a class can contain another class known as nested class. It's possible to create a nested class without giving any name.

A nested class that doesn't have any name is known as an anonymous class.

An anonymous class must be defined inside another class. Hence, it is also known as an anonymous inner class. Its syntax is:

```
class outerClass {
   // defining anonymous class
   object1 = new Type(parameterList) {
       // body of the anonymous class
   };
}
```

Note: Anonymous classes are defined inside an expression. So, the semicolon is used at the end of anonymous classes to indicate the end of the expression.

Anonymous classes usually extend subclasses or implement interfaces.

Here, Type can be

- a superclass that an anonymous class extends
- an interface that an anonymous class implements

The above code creates an object, object1, of an anonymous class at runtime.

```
class Polygon {
  public void display() {
    System.out.println("Inside the Polygon class");
  }
}

class AnonymousDemo {
  public void createClass() {

    // creation of anonymous class extending class Polygon
    Polygon p1 = new Polygon() {
      public void display() {
        System.out.println("Inside an anonymous class.");
      }
    };
    p1.display();
  }
}

class Main {
  public static void main(String[] args) {
     AnonymousDemo an = new AnonymousDemo();
     an.createClass();
  }
}
```

Advantages of Anonymous Classes

In anonymous classes, objects are created whenever they are required. That is, objects are created to perform some specific tasks. For example,

```
Object = new Example() {
  public void display() {
    System.out.println("Anonymous class overrides the method display().");
  }
};
```

Here, an object of the anonymous class is created dynamically when we need to override the display() method.

Anonymous classes also help us to make our code concise.

Java Singleton

Java Singleton ensures that only one object of a class can be created.

Here's how we can implement singletons in Java.

- create a private constructor that restricts to create an object outside of the class
- create a private attribute that refers to the singleton object.
- create a public static method that allows us to create and access the object we created. Inside the method, we will create a condition that restricts us from creating more than one object.

```
class Database {
  private static Database dbObject;
  private Database() {
  }
  public static Database getInstance() {
    // create object if it's not already created
    if(dbObject == null) {
      dbObject = new Database();
    }
     // returns the singleton object
     return dbObject;
  }
  public void getConnection() {
     System.out.println("You are now connected to the database.");
  }
}
class Main {
  public static void main(String[] args) {
    Database db1;

    // refers to the only object of Database
    db1= Database.getInstance();

    db1.getConnection();
  }
}
```

Singleton is a design pattern rather than a feature specific to Java. A design pattern is like our code library that includes various coding techniques shared by programmers around the world.

It's important to note that, there are only a few scenarios (like logging) where singletons make sense. Recommended that you avoid using singletons completely if you are not sure whether to use them or not.

Java enums

In Java, an enum (short for enumeration) is a type that has a fixed set of constant values. We use the enum keyword to declare enums. For example,

```
enum Size {
  SMALL, MEDIUM, LARGE, EXTRALARGE
}
```

Here, we have created an enum named Size. It contains fixed values SMALL, MEDIUM, LARGE, and EXTRALARGE.

These values inside the braces are called enum constants (values).

Note: The enum constants are usually represented in uppercase.

```
enum Size {
  SMALL, MEDIUM, LARGE, EXTRALARGE
}

class Main {
  public static void main(String[] args) {
    System.out.println(Size.SMALL);
    System.out.println(Size.MEDIUM);
  }
}
```

Also, we can create variables of enum types. For example,

```
Size pizzaSize;
```

Here, pizzaSize is a variable of the Size type. It can only be assigned with 4 values.

```
pizzaSize = Size.SMALL;
pizzaSize = Size.MEDIUM;
pizzaSize = Size.LARGE;
pizzaSize = Size.EXTRALARGE;
```

```
enum Size{
 SMALL, MEDIUM, LARGE, EXTRALARGE;
 public String getSize() {
  // this will refer to the object SMALL
  switch(this) {
   case SMALL:
    return "small";
   case MEDIUM:
    return "medium";
   case LARGE:
    return "large";
   case EXTRALARGE:
    return "extra large";
   default:
    return null;
   }
 }
 public static void main(String[] args) {
  // call getSize()
  // using the object SMALL
  System.out.println("The size of the pizza is " + Size.SMALL.getSize());
 }
}
```

Methods of Java Enum Class

There are some predefined methods in enum classes that are readily available for use.

1. Java Enum ordinal()

The ordinal() method returns the position of an enum constant. For example,

```
ordinal(SMALL)
// returns 0
```

2. Enum compareTo()

The compareTo() method compares the enum constants based on their ordinal value. For example,

```
Size.SMALL.compareTo(Size.MEDIUM)
 // returns ordinal(SMALL) - ordinal(MEDIUM)
```

3. Enum toString()

The toString() method returns the string representation of the enum constants. For example,

```
SMALL.toString()
// returns "SMALL"
```

4. Enum name()

The name() method returns the defined name of an enum constant in string form. The returned value from the name() method is final. For example,

```
name(SMALL)
// returns "SMALL"
```

5. Java Enum valueOf()

The valueOf() method takes a string and returns an enum constant having the same string name. For example,

```
Size.valueOf("SMALL")
// returns constant SMALL.
```

6. Enum values()

The values() method returns an array of enum type containing all the enum constants. For example,

```
Size[] enumArray = Size.value();
```

Why Java Enums?

In Java, enum was introduced to replace the use of int constants.
Suppose we have used a collection of int constants.

```
class Size {
  public final static int SMALL = 1;
  public final static int MEDIUM = 2;
  public final static int LARGE = 3;
  public final static int EXTRALARGE = 4;
}
```

Here, the problem arises if we print the constants. It is because only the number is printed which might not be helpful.
So, instead of using int constants, we can simply use enums. For example,

```
enum Size {
  SMALL, MEDIUM, LARGE, EXTRALARGE
}
```

This makes our code more intuitive.
Also, enum provides compile-time type safety.
If we declare a variable of the Size type. For example,
Size size;
Here, it is guaranteed that the variable will hold one of the four values. Now, If we try to pass values other than those four values, the compiler will generate an error.

Java enum Constructor

In Java, an enum class may include a constructor like a regular class. These enum constructors are either:

- **private** - accessible within the class

or

- **package-private** - accessible within the package

```
enum Size {
  // enum constants calling the enum constructors
  SMALL("The size is small."),
  MEDIUM("The size is medium."),
  LARGE("The size is large."),
  EXTRALARGE("The size is extra large.");
  private final String pizzaSize;
  // private enum constructor
  private Size(String pizzaSize) {
    this.pizzaSize = pizzaSize;
  }
  public String getSize() {
    return pizzaSize;
  }
}
class Main {
  public static void main(String[] args) {
    Size size = Size.SMALL;
    System.out.println(size.getSize());
  }
}
```

Java enum Strings

In Java, we can get the string representation of enum constants using the toString() method or the name() method. For example,

```
enum Size {
  SMALL, MEDIUM, LARGE, EXTRALARGE
}

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

    System.out.println("string value of SMALL is " + Size.SMALL.toString());
    System.out.println("string value of MEDIUM is " + Size.MEDIUM.name());

  }
}
```

Change Default String Value of enums

We can change the default string representation of enum constants by overriding the toString() method. For example,

```
enum Size {
  SMALL {
    // overriding toString() for SMALL
    public String toString() {
      return "The size is small.";
    }
  },
  MEDIUM {
    // overriding toString() for MEDIUM
    public String toString() {
      return "The size is medium.";
    }
  };
}
class Main {
  public static void main(String[] args) {
    System.out.println(Size.MEDIUM.toString());
  }
}
```

Note: We cannot override the name() method.
It is because the name() method is final.