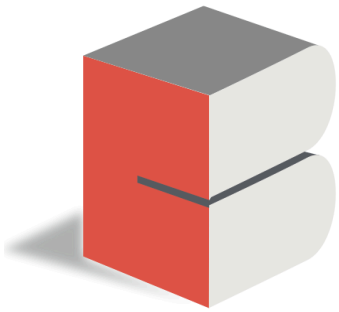


CODING BLOCKS

Crux
Object Oriented Programming

1. **Objects** are the real world entities about which we code. Objects have properties and they perform functions. For example in a student management system the real world entities about which the system revolves are – students, instructor, course, batch etc.

2. **A Class** is a template or a blue print and the objects are specific copies of it. For example a Vehicle class might look like :

```
public class Vehicle {
	public String brand;
	protected String model;
	private double price;
	int numWheels;
	int yearOfManufactor;
	String color;

	public double getPrice(){
		return price;
	}
	public  void printDescription(){
		System.out.println(brand +" " + model +"
	"+price+" "+numWheels);
	}
}
```

Now each vehicle will be a specific copy of this template. The syntax to create an object of vehicle is as follows :

```
public static void main(String args[]){
	Vehicle v = new Vehicle();
	//v.fieldName – will give access to this vehicle's
	// fields
}
```

Constructors – Constructor is a special method that is called when an object is instantiated i.e created. It is used to initialize an object. Its name is same as the class name. Even though in the above vehicle example we haven't created an explicit constructor there is a default constructor implicitly there. We can create our own constructor as well. Also we can define multiple constructors in a class as well. E.g :

```
public Vehicle(Double price){
      this.price = price;
}
```

Here this is a keyword that refers to current object, So this.price refers to the data member (i.e. price) of this object and not the argument variable price.
One important point to note here is that as soon as we create our constructor the default constructor goes off.

Now when we have defined the above constructor and if it is the only constructor in the class, then we can't create any object of Vehicle without giving its price. In a way we can actually restrict users that they can't create a vehicle without giving its price.

We can have more than one constructors within the same class (i.e constructor overloading), which constructor will be called will be decided on runtime depending on the type and number of arguments specified while creating the object.

3. Modifiers

1. **Static and Non-Static :** Static properties are those that belong to the class rather each specific object. So their separate copies aren't created. They are shared by all the objects of the class. You need to write static keyword before it in order to make it static.
   For e.g :

   ```
   static int numStudents;
   ```

   Here number of students in a batch is a property that isn't specific to each student and hence is static.
   Whereas properties like name, rollNumber etc can have different values for each object and are object specific and thus are non static.

2. **Access Modifiers**

   a) **Private :** It is visible only within the class i.e it can be accessed by and through the methods of the same class. So we can provide setters and getters function through which they can be accessed outside the class. For e.g the datafield price in the vehicle class shown above. So we can have getter and setter function for it .

   ```
   public double getPrice(){
         return price;
   }
   ```

```
public void setPrice(double price){
      if(price < 5000){
            return;
      }
      this.price = price;
}
```

b) **Default :** When we explicitly don't write any modifier it is default . This modifier is package friendly i.e it can be accessed within the same package.

c) **Protected :** It is accessible within the same package and outside the package but only through inheritance.

d) **Public :** It is accessible everywhere.

An important point to note here is that its better to make a variable private and then provide getters and setters in case we wish allow others to view and change it than making the variable public. Because by provding setter we can actually add constraints to the function and update value only if they are satisfied (say vehicle price can't be updated if its less than 5k).

3. **Final Keyword :** Final keyword can be applied before a variable, method and a class. A final variable is one whose value can't be changed. So we can either initialise a final variable at the time of declaration or in a constructor. A final method is one that can't be overriden. Making a class final means it can't be inherited. (E.g : String class in java is final)

4. **Abstract :** An abstract method is one which does not have implementation.
E.g

```
abstract void getType();
```

A class having even one abstract method has to be declared abstract, and since a abstract class is incomplete so you cannot create an instance of abstract class, but it can be extended. Also we can create reference of an abstract class. We will discuss more about in polymorphism.

Components Of OOPS

1. **Encapsulation -** Binding (or wrapping) code and data together into a single unit i.e a class is known as encapsulation. It lets us hide the implementation details using different access modifiers. Also it lets us change the implementation without breaking the code of users.

2. **Inheritance –** Inheritance is a mechanism by which one class can extend functionality from an existing class. It provides code reusability. The derived class inherits the states and behaviors from the base class. The derived class can add its own additional variables and methods. Syntax for inheritance is shown below –

```
public class Car extends Vehicle {
      private int numDoors;
      String company;

      public int numDoors(){
            return numDoors;
      }
}
```

Here car (sub class) extends Vehicle (base class / super class) since every car is a vehicle. So car will now have all the properties and functions that a vehicle has except the private fields of vehicle class(since they are not inherited , but they can be accessed via functions of base class that aren't private).

- What if both the base class and sub class have function with same signature i.e same name and arguments ? Say even car has a printDescription function as in vehicle.

```
public void printDescription(){
      System.out.println("Car :" + company +" " + model +"
"+getPrice()+" "+numDoors);
}
```

then

```
      Car c = new Car();
      c.printDescription();     // This will call car's printDescription
```

If we wish to call base class printDescription inside Car's printDescription then "super " keyword should be used.

```
super.printDescription();    // This will call Vehicle's printDescription
```

- Constructors – Suppose Vehicle has one constructor as shown below-

```
public Vehicle(Double price){
      this.price = price;
}
```

then Car needs to have a constructor that passes value to the vehicle constructor which is implicitly called when we create an object of car.

```
public Car(double price){
      super(price); // should be the first line
      numWheels = 4;
      company = "";
}
```

3. **Polymorphism –** It refers to one thing taking multiple forms. Following are different types of polymorphism that we should know about :

   3.1. *Ability of a variable to take different forms* – A base class' reference can refer to the object of its sub class i.e we can do something like this –

   ```
   Vehicle v = new Car(1000);
   ```

   Since every car is a vehicle so a vehicle(i.e. reference of type vehicle) can refer to a car. And not just the car, reference "v" here can refer to object of any other class that extends vehicle. But through this refernce "v" we can access only those properties of car which even a vehicle has i.e.

   ```
   v.numDoors = 4; // This will give error as numDoors is
         //    car's specific property
   ```

   3.2. *Overriding the base class functions(Virtual Functions)* – We have already seen its example above in inheritance. When both base class and sub class have functions of same signature then base class' function is overriden by the subclass' function.

   ```
   Vehicle v1 = new Vehicle();
   Vehicle v2 = new Car(1000);

   v1.printDescription(); // Will call vehicle's printDescription
   v2.printDescription(); // Will call car's printDescription
   ```

   In case of v2, which printDescription should be called is decided on runtime (Runtime Polymorphism) based on the type of the object and not the type of reference. Same is the case with abstract class, a reference of abstract class can refer to objects of all its sub classes which themselves aren't abstract.

3.3. Ability of a function to behave differently on basis of different parameters.

3.3.1. Function Overloading

```
public int add(int a,int b){
     return a+b;
}

public double add(double a,double b){
     return a+b;
}

public char add(char a,char b){
     return (char)(a+b);
}
```

Amongst these three add functions which add will be called finally, will be decided on runtime based on the type of parameters.

3.3.2. *Constructor Overloading*

Constructor overlaoding is similar to function overloading. At runtime while creating an object the number and type of parameters passed will decide that which constructor will be called.

```
public Vehicle(String color, double price){
     this.color = "white";
     this.price = price;
}

public Vehicle(double price){
     this.price = price;
}

public  Vehicle(){

}
```

3.4. Ability of a function to work with parameters of subtypes – This one is just an extension of first type.

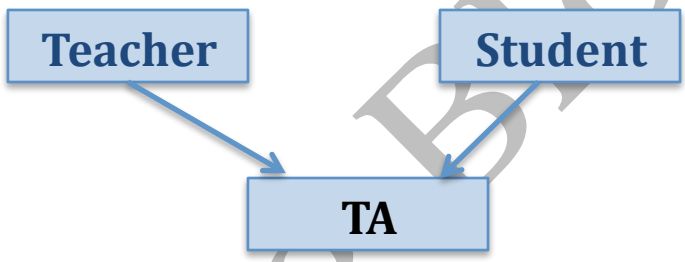
```
public void print(Vehicle v){
      v.printDescription();
}
```

This print function expects a vehicle, so we can pass a car(or object of any of its subtype) to it i.e.

```
Car c = new Car();
print(c);
```

Multiple Inheritance

When one class has more than one super classes then it is called multiple inheritance.
For e.g.



Here a TA is both a teacher and a student, it has two parent classes.
Java **doesn't allow** multiple inheritence, because it suffers from diamond problem. (https://en.wikipedia.org/wiki/Multiple_inheritance)

Interfaces

Interfaces in java are pure abstract i.e all methods in an interface are public and abstract and fileds are public, final and static by default. E.g :

```
public interface VehicleInterface {
      int a = 9;
      public String getType();
}
```

A class "implements" an interface. And a class implementing a interface must implement all its methods otherwise the class will have to be declared abstract.
Also a class can implement multiple interfaces and if it does so then it will have to implement all the methods in the interfaces that it is implementing.

```
public class Car extends Vehicle implements VehicleInterface{
	int numDoors;
	double mileage;

	public Car(double price){
		super(price);
		numWheels = 4;
		company = "";
	}

	@Override
	public String getType() {
		// TODO Auto-generated method stub
		return "Car";
	}
}
```

Thus interfaces serve as a contract, If a non abstract class is implemeting an interface then without looking into the code of the class we can be sure that the class must have implemented the methods in the interface, else the class would have been abstract.

Some points about interface :

1. We cannot instantiate an interface.
2. An interface does not contain any constructors.
3. All of the methods in an interface are abstract.
4. An interface cannot contain instance fields i.e non static fields. The only fields that can appear in an interface must be declared both static and final.
5. An interface is not extended by a class; it is implemented by a class.
6. An interface can extend multiple interfaces.

We can achieve runtime polymorphism in the same manner as discussed above for classes.

```
VehicleInterface v = new Car(1000);
System.out.println(v.getType());
System.out.println(v.a);
System.out.println(v.numDoors); // error
```

Generics

Suppose we make a pair class to store two ints.

```
public class Pair {
      int first;
      int second;
}
```

Now if we want to have a pair of two chars/strings/double then we will have t create separate pair classes for each of them. Generics allow us to create a single Pair class that will work for different types.

Creating a generic class

Syntax of a generic Pair class is :

```
public class Pair<T> {
      T first;
      T second;
}
```

Here <T> represents the type parameter and "T" is an identifier that specifies a generic type name, it could have been any other letter.

Now if we want to have pair of two ints, then syntax will be :

```
Pair<Integer> pInts = new Pair<Integer>();
Pair<String> pStrings = new Pair<Strings>(); // Pair of two
                                             // Strings
```

Note that type parameters can represent only reference types, not primitive types. So for primitives int,char etc java has corressponding Wrapper classes Integer, Character etc.

A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety

Multiple Type Parameters

We can have multiple type parameters as well i.e we can create a pair class where "first" and "second" can be of different types unlike the Pair class defined above where both have to be of same type.

```
public class Pair<T,S> {
      T first;
      S second;
}
```

Its instance can be created as follows :

```
Pair<Integer,String> pair = new Pair<Integer>();
```

So here pair.first is a Integer and pair.second is a String.

Multilayer Generic Parameters

The genric parameters can be multiplayered.

```
Pair<Pair<Integer>> pLayered = new Pair<>();
```

Here pLayered.first and pLayered.second are themselves pair of Integers.
Similarly we can add multiple layers to the parameters.

Generic Methods

Just like we saw generic classes above, we can make methods also accept generic parameters. We can make a method generic even when the class isn't generic. Here is a sample printArray method that works for generic inputs.

```
public static<T> void printArray(T input[]){
     for(int i = 0; i < input.length; i++){
          System.out.print(input[i] +" ");
     }
}
```

Bounded Type Parameters:

Many a times when you might want to restrict the kinds of types that are allowed to be passed to a type parameter. Say we want to create a generic sort function.

In order to sort elements we will have to compare them. The "<" or ">" are not defined for non–primitives. So instead we will have to use compareTo() method (in Comparable interface) which compares two objects and returns an int based on result.

Now in our sort function we should allow only those non-primitives who have compareTo() method defined for them or in other terms who have implemented the Comparable interface (as interface serves as a contract, so if a non-abstract class has implemeted has implemented Comparable method then we can be sure that it has compareTo() method). We can do this as shown below :

```
public static<T extends Comparable<T>> void sort(T input[]){
          T temp;
          for(int i = 0; i < input.length; i++){
               for(int j = 0; j < input.length - i - 1; j++){
                    if(input[j].compareTo(input[j+1])==1){
                         temp = input[j+1];
                         input[j+1] = input[j];
                         input[j] = temp;
                    }
               }
          }
     }
```

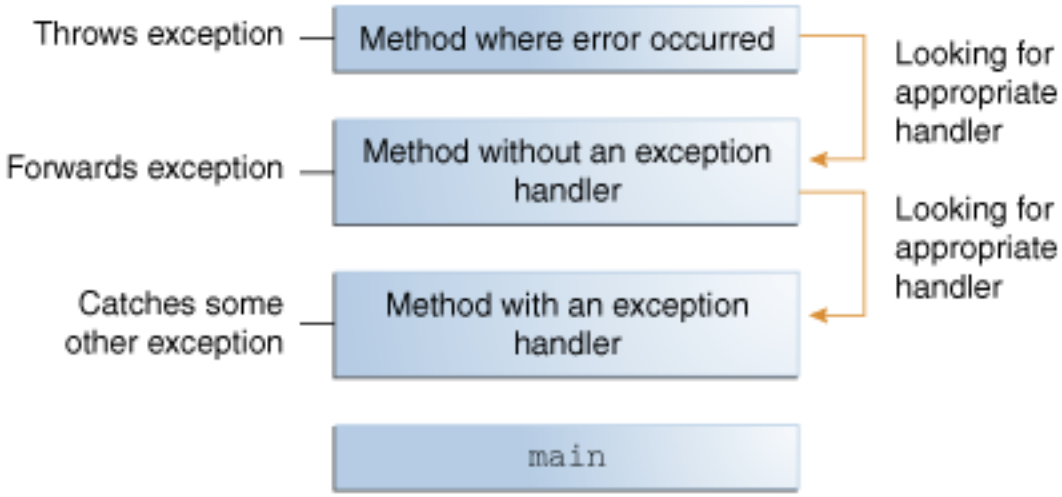
When we write <T extends Comparable<T>> this means that only those parameters are allowed those who have implemented Comparable Interface.

Exceptions

An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions. The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called throwing an exception.

After a method throws an exception, the runtime system attempts to find something to handle it. The block of code that handles an exception is called exception handler. When an exception occurs the run time system first tries to find an exception handler in the method where the exception occurred and then searches the methods in the reverse order in which they were called for the exception handler. The list of methods is known as the call stack(shown below). If no method handles the exception then exception appears on the console (like we see ArrayIndexOutOfBoundsException etc)



Types of Exception

1. **Checked Exceptions :** These are exceptional conditions that we can anticipate when user makes mistake . For example computing factorial of a negative number. A well-written program should catch this exception and notify the user of the mistake, possibly prompting for a correct input. Checked exceptions *are subject* to the Catch or Specify Requirement i.e either the function where exception can occur should handle or specify that it can throw an exception (We will look into it in detail later).
2. **Error :** These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from. For example, suppose that an application successfully opens a file for input, but is unable to read the file because of a hardware or system malfunction.
3. **Unchecked Exception :** These are exceptional conditions that might occur at runtime but we don't expect them to occur while writing code. These usually indicate programming bugs, such as logic errors or improper use of an API. For example StackOverflowException.

Exception Handling

Exception handling is achieved by using try catch and/or finally block.

Try block - The code which can cause an exception is enclosed within try block.

Catch block - The action to be taken when an exception has occurred is done in catch block. It must be used after the try block only.

Finally block - Java finally block is a block that is used to execute important code such as closing connection, stream etc. Java finally block is always executed whether exception is handled or not.

Here is a sample code to explain the same.

```
public static void main(String[] args){
      Scanner s = new Scanner(System.in);
      System.out.println("Enter dividend ");
      int dividend = s.nextInt();
      System.out.println("Enter divisor ");
      int divisor = s.nextInt();
      try{
            int data= dividend/divisor;
            System.out.println(data);
      }
      catch(ArithmeticException e){
            System.out.println(“Divide by zero error”);
      }
      finally{
            System.out.println("finally block is always
            executed");
      }
      System.out.println("rest of the code...");
}
```

Note :

1. Whenever an exception occurs statements in the try block after the statement in which exception occurred are not executed
2. For each try block there can be zero or more catch blocks, but only one finally block.

Creating an Exception / User Defined Exceptions

A user defined exception is a sub class of the exception class. For creating an exception you simply need to extend Exception class as shown below :

```
public class InvalidInputException extends Exception {
      private static final long serialVersionUID = 1L;
}
```

Throwing an Exception

Sometimes, it's appropriate for code to catch exceptions that can occur within it. In other cases, however, it's better to let a method further up the call stack handle the exception. For example if input to the factorial method is a negative number, then it makes more sense for the factorial to throw an exception and the method that has called factorial method to handle the exception.

Here is the code for the factorial method :

```
public static int fact(int n) throws InvalidInputException{
	if(n < 0){
		InvalidInputException e = new InvalidInputException();
		throw e;
	}
	if(n == 0){
		return 1;
	}
	return n*fact(n-1);
}
```

The fact method throws an InvalidInputException that we created above and we will handle the exception in main.

```
public static void main(String[] args)  {
	Scanner s = new Scanner(System.in);
	System.out.println("Enter number ");
	int n = s.nextInt();
	int a = 10;
	try{
	System.out.println(fact(n));
		a++;
	}
	catch(InvalidInputException e){
		System.out.println("Invalid Input !! Try again");
		return;
	}
}
```