Java

Unit 1

Class

- template / blueprint for creating obj (instance of class).
- defines attributes and behaviours of obj.
- Ex. Car:
    - attr: make, model, color, number of doors.
    - behaviour: starting of engine, accelerating, braking.

Object

- specific instance of a class
- created from class
- has its own set of values for attrs.
- Ex: Toyota model, color blue, no. of doors 4.

Evolution of Java, Byte Code, JDK, JVM, JRE

Here is an explanation in points for the evolution of Java, byte code, JDK, JVM, and JRE:

1. In 1991, James Gosling and his team at Sun Microsystems started the development of a new programming language called Oak, which later became Java.
2. The first version of Java, called Java 1.0, was released in 1996.
3. Java introduced the concept of write once, run anywhere (WORA), which means that code written in Java can run on any platform that has a Java Virtual Machine (JVM) installed.
4. Java programs are compiled into bytecode, which is an intermediate language that can be executed by a JVM.
5. The JVM is a virtual machine that interprets the bytecode and executes it on the underlying operating system.
6. The Java Development Kit (JDK) is a software development kit that includes tools for developing, compiling, and debugging Java programs.
7. The JDK includes the Java Runtime Environment (JRE), which is a subset of the JDK and includes the JVM and essential libraries for running Java programs.
8. The JRE is installed on end-user machines to run Java applications.
9. As Java evolved, new versions were released, adding new features and improving performance.
10. In 2010, Oracle Corporation acquired Sun Microsystems and became the steward of the Java platform.

11. Today, Java is widely used for developing a variety of applications, including web applications, desktop applications, mobile applications, and enterprise software.

Application and benefits of OOPs

An application is a program or software that performs a specific task or set of tasks. Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of objects, which are instances of classes that contain data and methods.

Some of the benefits of using OOP in software development include:

1. Modularity: OOP allows you to break down complex systems into smaller, more manageable modules (objects) that can be reused in different parts of your application or in other applications altogether.
2. Encapsulation: By bundling data and behavior within a single object, OOP promotes encapsulation, which protects the data from outside interference and ensures that each object is responsible for its own behavior.
3. Inheritance: In OOP, classes can inherit properties and behavior from other classes. This enables developers to create new classes that extend or modify the behavior of existing classes, thereby reducing redundancy and improving code reuse.
4. Polymorphism: OOP supports polymorphism, which allows developers to create multiple implementations of the same method, each tailored to a specific class or situation. This can make your code more flexible and adaptable to changing requirements.
5. Abstraction: OOP promotes abstraction, which allows developers to focus on the essential characteristics of an object and ignore the details that are irrelevant to its behavior. This can make your code more understandable and maintainable.

Overall, OOP provides a powerful set of tools for software development that can help you build more modular, reusable, and maintainable applications.

Encapsulation

Make sure that "sensitive" data is hidden from users. To achieve this: - declare class variables/attributes as private. - provide public get and set methods to access and update the value of a private variable.

Inheritance

- inherit attr. and methods from one class to another.
- 2 categories:
  - subclass (child)
  - superclass (parent)
- uses extends keyword

```
class Vehicle {
  protected String brand = "Ford";        // Vehicle attribute
  public void honk() {                    // Vehicle method
    System.out.println("Tuut, tuut!");
```

```
  }
}

class Car extends Vehicle {
  private String modelName = "Mustang";    // Car attribute
  public static void main(String[] args) {

    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (from the Vehicle class) on the myCar object
    myCar.honk();

    // Display the value of the brand attribute (from the Vehicle class) and the
value of the modelName from the Car class
    System.out.println(myCar.brand + " " + myCar.modelName);
  }
}
```

Multilevel vs multiple inheritance

Multilevel

- ability of a subclass to inherit from a superclass that itself inherits from another superclass.
- each subclass only one direct superclass.

```
public class Animal {
    public void eat() {
        System.out.println("The animal is eating.");
    }
}

public class Mammal extends Animal {
    public void sleep() {
        System.out.println("The mammal is sleeping.");
    }
}

public class Dog extends Mammal {
    public void bark() {
        System.out.println("The dog is barking.");
    }
}

public class InheritanceExample {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat(); // output: The animal is eating.
        d.sleep(); // output: The mammal is sleeping.
        d.bark(); // output: The dog is barking.
```

```
    }
}
```

Multiple Inheritance

- ability of a subclass to inherit from multiple superclasses.
- subclass can inherit properties and methods from more than one superclass
- not all programming languages support multiple inheritance, as it can lead to conflicts and ambiguity when two or more superclasses have the same method or property names.

```
public class Employee {
    private String name;
    private int age;

    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void printInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}

public class Manager {
    private String department;

    public Manager(String department) {
        this.department = department;
    }

    public void printDepartment() {
        System.out.println("Department: " + department);
    }
}

public class ManagerEmployee extends Employee, Manager {
    private String title;

    public ManagerEmployee(String name, int age, String department, String title)
{
        super(name, age);
        this.department = department;
        this.title = title;
    }

    public void printTitle() {
        System.out.println("Title: " + title);
    }
}
```

```
public class InheritanceExample {
    public static void main(String[] args) {
        ManagerEmployee me = new ManagerEmployee("John Doe", 35, "Sales", "Sales
Manager");
        me.printInfo(); // output: Name: John Doe Age: 35
        me.printDepartment(); // output: Department: Sales
        me.printTitle(); // output: Title: Sales Manager
    }
}
```

Polymorphism

- "many forms"
- uses **inherited** methods to perform diff. tasks.

```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}
```

Runtime vs Compiletime polymorphism

1. Runtime:

- dynamic polymorphism
- overriding
- decision about which implementation of the method to execute is made at runtime, based on actual object.

```
public class Animal {
    public void makeSound() {
        System.out.println("The animal makes a sound.");
    }
}
```

```
public class Dog extends Animal {
    public void makeSound() {
        System.out.println("The dog barks.");
    }
}

public class Cat extends Animal {
    public void makeSound() {
        System.out.println("The cat meows.");
    }
}

public class PolymorphismExample {
    public static void main(String[] args) {
        Animal a1 = new Animal();
        Animal a2 = new Dog();
        Animal a3 = new Cat();

        a1.makeSound(); // output: The animal makes a sound.
        a2.makeSound(); // output: The dog barks.
        a3.makeSound(); // output: The cat meows.
    }
}
```

1. Compiletime:

- static polymorphism
- method overloading
- compiler decides which implementation of a method to execute at compile time, based on the number and types of arguments that are passed to the method.

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

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

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

public class PolymorphismExample {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        int sum1 = calc.add(2, 3); // calls add(int a, int b)
        double sum2 = calc.add(2.5, 3.5); // calls add(double a, double b)
```

```
        String sum3 = calc.add("Hello", " world!"); // calls add(String a, String
b)
        System.out.println(sum1); // output: 5
        System.out.println(sum2); // output: 6.0
        System.out.println(sum3); // output: Hello world!
    }
}
```

Variables

- Local Variables:
  - declared inside a method, constructor, or block
  - can only be accessed within that scope
  - created when the method is called and destroyed when the method returns

```
public void exampleMethod() {
   int x = 5; // local variable
   System.out.println(x);
}
```

- instance variable
  - declared within a class but outside of any method
  - created when an object of the class is created
  - accessed by any method within the class and have a separate value for each object of the class

```
public class ExampleClass {
   int y; // instance variable

   public void setY(int newValue) {
      y = newValue;
   }

   public void printY() {
      System.out.println(y);
   }
}
```

- class variable
  - static variables
  - declared within a class but outside of any method (same as instance var)
  - BUT shared by all objects of the class
  - accessed by any method within the class and can also be accessed directly using the class name

```
public class ExampleClass {
   static int z; // class variable
```

```
    public void setZ(int newValue) {
        z = newValue;
    }

    public void printZ() {
        System.out.println(z);
    }
}
```

Contructor chaining

- avoid duplicating the code in the constructors and make the code more concise and maintainable

```
public class Person {
    private String name;
    private int age;

    public Person() {
        this("Unknown", 0);
    }

    public Person(String name) {
        this(name, 0);
    }

    public Person(int age) {
        this("Unknown", age);
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

this and super

- keywords used to refer to different objects.
- `this` refers to the current object, or the object that is executing the current method. It is often used to differentiate between instance variables and local variables with the same name.
- `super` refers to the parent class of the current class. It is often used to call methods or access instance variables defined in the parent class.

```
public class Person {
    private String name;

    public Person(String name) {
```

```
        this.name = name; // "this" refers to the instance variable "name"
    }
}

public class Student extends Person {
    private int grade;

    public Student(String name, int grade) {
        super(name); // call the constructor of the parent class with the "name"
parameter
        this.grade = grade;
    }

    public void display() {
        super.display(); // call the "display" method of the parent class
        System.out.println("Grade: " + grade);
    }
}
```

Abstract class in java in points

Here are some points about abstract class in Java:

1. Abstract class is a class that contains at least one abstract method, which is declared but not defined in the abstract class.
2. An abstract class can contain both abstract and non-abstract (concrete) methods.
3. Abstract classes can have constructors and instance variables.
4. Abstract classes cannot be instantiated; they can only be subclassed (i.e., extended) to create concrete classes.
5. Subclasses of an abstract class must either provide an implementation for all the abstract methods in the superclass or themselves be declared as abstract.
6. Abstract classes can be used to define a common interface or set of methods for a group of related classes.
7. Abstract classes are useful for providing a base implementation that can be customized by subclasses.
8. Abstract classes are often used in template method design pattern.
9. Abstract classes can be used to implement interfaces partially.
10. Java allows a class to extend only one abstract class but it can implement multiple interfaces.

abstract example

Sure, here's an example of an abstract class in Java and how to extend it:

```java
// abstract class
abstract class Shape {
  // abstract method
  public abstract double getArea();
}

// concrete subclass extending Shape
class Rectangle extends Shape {
  private double length;
  private double width;

  public Rectangle(double length, double width) {
    this.length = length;
    this.width = width;
  }

  public double getArea() {
    return length * width;
  }
}

// usage
public class Main {
  public static void main(String[] args) {
    Shape shape = new Rectangle(5.0, 3.0);
    System.out.println("Area of rectangle: " + shape.getArea());
  }
}
```

In this example, `Shape` is an abstract class with an abstract method `getArea()`. `Rectangle` is a concrete subclass of `Shape` that implements `getArea()` to return the area of the rectangle.

In the `main` method, we create a new `Rectangle` object with a length of 5.0 and a width of 3.0 and assign it to a variable of type `Shape`. We can then call the `getArea()` method on the `Shape` object, which actually invokes the implementation in the `Rectangle` subclass and returns the area of the rectangle.

Invalid combo

- final abstract
- private abstract
- static abstract

interface

An interface in Java is a collection of abstract methods and constants. Here are some key points regarding interfaces in Java:

1. An interface is declared using the `interface` keyword.
2. An interface can contain only abstract methods, static and default methods, and constants.

3. An interface cannot have any constructors.
4. All the methods declared in an interface are by default public and abstract.
5. An interface can be implemented by a class using the `implements` keyword.
6. A class can implement multiple interfaces.
7. Implementing an interface is a way to achieve multiple inheritances in Java.
8. An interface can also extend another interface using the `extends` keyword.
9. A class implementing an interface must implement all the methods declared in that interface.
10. An interface is used to provide a common behavior to unrelated classes.
11. The primary use of interfaces in Java is to implement abstraction.
12. Interfaces provide a way to achieve loose coupling between objects.
13. An interface can be used to define a contract that a class must follow in order to be used in a particular way.
14. Interfaces can be used to define a set of related methods that can be implemented by different classes in different ways, providing polymorphism.
15. The use of interfaces in Java makes it easy to replace or modify the implementation of a class without affecting the code that uses it.

interface stuff

```
interface A {
    int a = 10;
    void uwu();
}
```

javap Car

```
Compiled from "Car.java"
interface A {
  public static final int a;
  public abstract void uwu();
}
```

Multiple inhereitance via interface with example

In Java, multiple inheritance is not allowed through classes. However, multiple inheritance can be achieved through interfaces. An interface can extend multiple interfaces, allowing a class that implements the interface to inherit from all the extended interfaces.

Here's an example of multiple inheritance using interfaces in Java:

```
interface InterfaceA {
    void methodA();
}

interface InterfaceB {
    void methodB();
}

interface InterfaceC extends InterfaceA, InterfaceB {
    void methodC();
}

class MyClass implements InterfaceC {
    public void methodA() {
        System.out.println("Method A");
    }

    public void methodB() {
        System.out.println("Method B");
    }

    public void methodC() {
        System.out.println("Method C");
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.methodA();
        obj.methodB();
        obj.methodC();
    }
}
```

In the above example, we have three interfaces - `InterfaceA`, `InterfaceB`, and `InterfaceC`. `InterfaceC` extends both `InterfaceA` and `InterfaceB`. The `MyClass` class implements `InterfaceC` and provides the implementation for all three methods - `methodA()`, `methodB()`, and `methodC()`.

When we run the program, it will output:

```
Method A
Method B
Method C
```

This shows that we have successfully achieved multiple inheritance through interfaces.

extends vs implements

In Java, `extends` and `implements` are both keywords used to define inheritance, but they are used in different contexts.

`extends` is used to define inheritance between classes. When one class `extends` another class, it inherits all the non-private properties and methods of the superclass. The subclass can also add new properties and methods, and override the methods of the superclass. Example:

```
public class Animal {
    public void eat() {
        System.out.println("Eating...");
    }
}

public class Dog extends Animal {
    public void bark() {
        System.out.println("Woof!");
    }
}

Dog dog = new Dog();
dog.eat();  // Output: Eating...
dog.bark(); // Output: Woof!
```

`implements` is used to define inheritance between a class and an interface. When a class `implements` an interface, it must provide an implementation for all the methods defined in the interface. Example:

```
public interface Vehicle {
    void drive();
}

public class Car implements Vehicle {
    public void drive() {
        System.out.println("Driving a car...");
    }
}

Car car = new Car();
car.drive(); // Output: Driving a car...
```

In summary, `extends` is used for inheritance between classes, while `implements` is used for inheritance between a class and an interface.

explain anonymous class , also give example.

In Java, an anonymous class is a local class without a name. It is used when you need to create a class that will only be used once and you don't want to create a separate class for it. Anonymous classes are created and

instantiated at the same time.

Here is an example of an anonymous class:

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

       // Creating an instance of an anonymous class that implements Runnable
interface
       Runnable r = new Runnable() {
           public void run() {
               System.out.println("Hello from anonymous class!");
           }
       };

       // Starting the thread
       Thread t = new Thread(r);
       t.start();
   }
}
```

In this example, we create an anonymous class that implements the Runnable interface. We override the `run()` method to print a message. We then create a Thread object and pass the anonymous class as a parameter. We then start the thread to execute the `run()` method, which will print the message "Hello from anonymous class!" to the console.

Upcasting and downcasting

In Java, upcasting and downcasting are used to convert between related reference types. Upcasting is converting a reference of a subclass type to a reference of a superclass type, while downcasting is converting a reference of a superclass type to a reference of a subclass type. Here's an example to illustrate both upcasting and downcasting:

```
class Animal {
  public void eat() {
    System.out.println("Animal is eating.");
  }
}

class Dog extends Animal {
  public void bark() {
    System.out.println("Dog is barking.");
  }
}

public class Main {
  public static void main(String[] args) {
    // upcasting
    Animal animal = new Dog();
```

```
        animal.eat(); // output: Animal is eating.
        // animal.bark(); // error: bark() method is not available on Animal type

        // downcasting
        Dog dog = (Dog) animal;
        dog.bark(); // output: Dog is barking.
    }
}
```

In this example, `Animal` is a superclass of `Dog`. `Dog` inherits the `eat()` method from `Animal` and adds its own method `bark()`. In the `main` method, we create a new `Dog` object and assign it to a variable of type `Animal`. This is an example of upcasting, where a subclass object is assigned to a variable of its superclass type.

Even though the object is of type `Dog`, we can only access the methods and variables of the `Animal` class using the `animal` reference. In this case, we can call the `eat()` method, but not the `bark()` method because `animal` is of type `Animal`.

Next, we perform downcasting by casting the `animal` reference to `Dog` type using `(Dog) animal`. This is possible because we know that `animal` is actually referencing a `Dog` object. Now, we can use the `dog` reference to access the `bark()` method.

It's important to note that downcasting can be unsafe if the object being casted is not actually an instance of the target subclass type. In such cases, a `ClassCastException` will be thrown at runtime. To prevent this, it's recommended to use the `instanceof` operator to check the object's type before performing downcasting.

Array

Here are some key points about arrays in Java:

1. An array is a collection of elements of the same data type.
2. Arrays have a fixed length that is defined when they are created.
3. The elements of an array are stored in contiguous memory locations.
4. Arrays are objects in Java and are dynamically allocated on the heap.
5. Elements of an array can be accessed using an index, starting at 0 for the first element.
6. Java supports multidimensional arrays, where each element is itself an array.
7. Java arrays can be initialized with values at the time of creation or can be assigned values later.
8. The length of an array can be obtained using the `length` field.
9. Java provides various methods to manipulate arrays such as sorting, searching, and copying.
10. Arrays in Java are passed by reference, meaning a reference to the array is passed to a method instead of a copy of the array.

Overall, arrays are a fundamental data structure in Java that provide a convenient way to store and manipulate collections of elements.

array stuff

java wrapper class in points with example

Wrapper classes in Java are used to represent primitive data types as objects. They provide a way to use primitive data types as objects, and they also provide utility methods to perform certain operations on these

objects.

Some of the commonly used wrapper classes in Java are:

1. Integer: used to wrap int values
2. Double: used to wrap double values
3. Character: used to wrap char values
4. Boolean: used to wrap boolean values

Example:

```
// Wrapping an int value in an Integer object
Integer num = new Integer(10);

// Using utility methods of Integer class
System.out.println(num.intValue()); // prints 10
System.out.println(Integer.parseInt("20")); // prints 20

// Unwrapping Integer object to int value
int x = num.intValue();
```

In the above example, we are creating an Integer object to wrap an int value, and then using the `intValue()` method to get the int value back. We are also using the `parseInt()` method of the Integer class to convert a String to an int value.

string buffer

StringBuffer in Java is a class that provides a mutable sequence of characters. It is similar to the String class, but unlike Strings, StringBuffers can be modified. StringBuffer is thread-safe which means multiple threads cannot access it simultaneously. Here are some important points about StringBuffer:

1. StringBuffer is a mutable sequence of characters.
2. It is similar to the String class, but unlike Strings, StringBuffer can be modified.
3. StringBuffer is thread-safe which means multiple threads cannot access it simultaneously.
4. StringBuffer provides methods to append, insert, delete, and replace characters.
5. StringBuffer is efficient when it comes to modifying a string, especially when compared to creating a new string object.
6. StringBuffer is used when you need to modify strings frequently or when you are dealing with large strings.

Here is an example of how to use StringBuffer to modify a string:

```
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World");
System.out.println(sb.toString()); // Output: "Hello World"
sb.insert(6, ", ");
System.out.println(sb.toString()); // Output: "Hello, World"
sb.delete(5, 7);
System.out.println(sb.toString()); // Output: "HelloWorld"
```

```
sb.replace(5, 8, " everyone");
System.out.println(sb.toString()); // Output: "Hello everyone"
```

import in java

In Java, `import` is a keyword used to import a package or a specific class into the current file or scope, allowing access to its public members (fields, methods, and nested classes).

Here are some key points about `import` in Java:

- `import` statements are placed at the top of the Java file, after the package declaration (if any) and before the class declaration.
- They specify the fully qualified name of the package or class to be imported, and can include wildcards (`*`) to import all public members of a package or subpackage.
- Multiple `import` statements can be included in a file to import from multiple packages or classes.
- If the imported class or package name conflicts with a class or package in the current scope, the imported name can be aliased using the `as` keyword.

Here is an example of importing a class in Java:

```
import java.util.ArrayList;

public class MyClass {
    public static void main(String[] args) {
        ArrayList<String> myList = new ArrayList<>();
        // Do something with the ArrayList...
    }
}
```

In this example, the `ArrayList` class from the `java.util` package is imported using the `import` statement, which allows it to be used directly in the code without the need for a fully qualified name.

static import in java

In Java, the `import` statement is used to bring a class or a package into scope so that it can be used in the current program. The `static import` statement is a variant of the `import` statement that allows static members of a class to be imported directly into the current class without using the class name.

When you use `static import`, you can access the static members of a class directly without qualifying them with the class name. This makes the code more concise and readable.

Here is an example of using `static import` to access the `PI` constant of the `Math` class directly:

```
import static java.lang.Math.PI;

public class Circle {
   public double getArea(double radius) {
      return PI * radius * radius;
```

```
    }
}
```

In this example, we have imported the `PI` constant of the `Math` class using the `static import` statement. This allows us to use `PI` directly in the `getArea` method of the `Circle` class without qualifying it with the class name `Math`.

explain package in java in points with example

Packages in Java are a way of organizing related classes and interfaces together. It provides a mechanism for grouping related classes/interfaces, which makes it easier to locate and use them. Here are some points that explain packages in Java:

1. Packages in Java help to avoid naming conflicts that may occur when we have multiple classes with the same name.
2. Packages are hierarchical in nature, i.e., they can have sub-packages within them.
3. The primary use of packages is to group related classes/interfaces and provide access protection.
4. The access modifiers in Java (public, protected, private) can be used with packages to control the visibility of classes/interfaces within the package.
5. Packages can be imported using the "import" keyword in Java. This helps to avoid fully qualified class names in the code.
6. A package is declared using the "package" keyword in Java.

Example:

Consider a scenario where we have multiple classes that deal with database operations. We can group all of these classes into a package named "com.example.db". Here, "com" represents the organization that created this package, "example" represents the project name, and "db" represents the purpose of the package.

To declare a package, we can use the following statement at the beginning of each class:

```
package com.example.db;
```

To use a class from this package in another class, we can import it using the "import" statement:

```
import com.example.db.MyClass;
```

Here, "MyClass" is a class defined in the "com.example.db" package.

exception handling

- use throw for custom exception

```
public class Main {
  static void checkAge(int age) {
    if (age < 18) {
      throw new ArithmeticException("Access denied - You must be at least 18 years
old.");
    }
    else {
      System.out.println("Access granted - You are old enough!");
    }
  }

  public static void main(String[] args) {
    checkAge(15); // Set age to 15 (which is below 18...)
  }
}
```

exceptions

checked and unchecked exceptions

In Java, exceptions are used to handle errors or exceptional situations that may occur during program execution. There are two types of exceptions in Java: checked and unchecked exceptions.

1. Checked exceptions: Checked exceptions are the exceptions that are checked by the compiler at the compile-time. If a method throws a checked exception, then the calling method is required to handle it, either by catching it or by declaring it in its throws clause. Some examples of checked exceptions in Java are IOException, ClassNotFoundException, and SQLException.

2. Unchecked exceptions: Unchecked exceptions are the exceptions that are not checked by the compiler at the compile-time. They are thrown at runtime when an exceptional situation occurs. Unlike checked exceptions, the calling method is not required to handle unchecked exceptions. Some examples of unchecked exceptions in Java are NullPointerException, ArrayIndexOutOfBoundsException, and ArithmeticException.

It is a good practice to use checked exceptions for recoverable errors, such as input/output errors or database errors, and unchecked exceptions for unrecoverable errors, such as programming errors or runtime errors.

Here is an example of how to handle checked and unchecked exceptions in Java:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class ExceptionExample {
    public static void main(String[] args) {
        // Example of a checked exception (FileNotFoundException)
        try {
            File file = new File("input.txt");
            Scanner scanner = new Scanner(file);
```

```
            while (scanner.hasNextLine()) {
                String line = scanner.nextLine();
                System.out.println(line);
            }
            scanner.close();
        } catch (FileNotFoundException e) {
            System.out.println("File not found.");
            e.printStackTrace();
        }

        // Example of an unchecked exception (ArithmeticException)
        int a = 10;
        int b = 0;
        try {
            int result = a / b;
            System.out.println(result);
        } catch (ArithmeticException e) {
            System.out.println("Divide by zero error.");
            e.printStackTrace();
        }
    }
}
```

In this example, we handle a checked exception (FileNotFoundException) by catching it and printing the error message using the printStackTrace() method. We also handle an unchecked exception (ArithmeticException) by catching it and printing the error message.

list runtime and compiletime exceptions

In Java, there are two types of exceptions: checked exceptions and unchecked exceptions.

Checked Exceptions Checked exceptions are the exceptions that are checked at compile-time by the compiler. They occur during the execution of the program and may be recoverable. Some of the common checked exceptions are:

- IOException
- SQLException
- ClassNotFoundException
- InvocationTargetException
- ParseException

Unchecked Exceptions Unchecked exceptions are the exceptions that are not checked at compile-time by the compiler. They occur during the execution of the program and are usually caused by programming errors or bugs. Some of the common unchecked exceptions are:

- NullPointerException
- ArrayIndexOutOfBoundsException
- ClassCastException
- IllegalArgumentException
- ArithmeticException

It is important to handle exceptions properly in order to avoid unexpected crashes or behavior of the program.

explain Object class in java in points with example

Sure, here are the key points about the Object class in Java with an example:

1. Object class is the root class of all Java classes and is located in the java.lang package.
2. Every class in Java implicitly extends the Object class.
3. Object class has several methods like toString(), equals(), hashCode(), etc., which are inherited by all Java classes.
4. The toString() method returns a string representation of the object and is used to print the object.
5. The equals() method compares two objects for equality and returns a boolean value.
6. The hashCode() method returns an integer hash code value for the object.
7. Here's an example of how the Object class is implicitly used in Java:

```
public class Employee {
    private int id;
    private String name;

    public Employee(int id, String name) {
        this.id = id;
        this.name = name;
    }

    // getters and setters

    @Override
    public String toString() {
        return "Employee [id=" + id + ", name=" + name + "]";
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Employee other = (Employee) obj;
        if (id != other.id)
            return false;
        if (name == null) {
            if (other.name != null)
                return false;
        } else if (!name.equals(other.name))
            return false;
        return true;
    }
```

```
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + id;
        result = prime * result + ((name == null) ? 0 : name.hashCode());
        return result;
    }
}
```

In the above example, the Employee class implicitly extends the Object class, and it overrides the toString(), equals(), and hashCode() methods.

array copy java

In Java, you can use the `System.arraycopy()` method to copy an array. This method allows you to copy a range of elements from one array to another array.

The syntax of the `System.arraycopy()` method is as follows:

```
public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
```

- `src`: The source array.
- `srcPos`: The starting position in the source array.
- `dest`: The destination array.
- `destPos`: The starting position in the destination array.
- `length`: The number of elements to copy.

Here is an example program that demonstrates how to use the `System.arraycopy()` method:

```
public class ArrayCopyExample {
    public static void main(String[] args) {
        int[] source = {1, 2, 3, 4, 5};
        int[] destination = new int[5];

        System.arraycopy(source, 0, destination, 0, source.length);

        // Print the elements of the destination array
        for (int i = 0; i < destination.length; i++) {
            System.out.print(destination[i] + " ");
        }
    }
}
```

In this example, we create a source array `source` with five elements and a destination array `destination` with the same size. We then use the `System.arraycopy()` method to copy the entire `source` array to the

`destination` array. Finally, we print the elements of the `destination` array to verify that the copy was successful.

boxing and autoboxing

Boxing is the process of converting a primitive data type to its corresponding object wrapper class, and autoboxing is the automatic conversion of primitive data types to their corresponding object wrapper classes by the Java compiler.

```
// Boxing
double d = 3.14;
Double dObj = Double.valueOf(d);

// Autoboxing
char c = 'A';
Character cObj = c;

// Unboxing
Long longObj = Long.valueOf(123456789);
long l = longObj.longValue();

// Autounboxing
Boolean boolObj = Boolean.valueOf(true);
boolean bool = boolObj;
```