Explain the difference between primitive and reference data types with examples.

Answer: Differences between Primitive and Reference Data Types

Feature	Primitive Data Type	Reference Data Type
Storage	Stores the actual value	Stores the memory reference
Location	Stored in stack memory	Stored in heap memory
Size	Fixed size	Size depends on the object
Nullability	Cannot be null	Can be null
Inheritance	Cannot be inherited	Can be inherited (since objects are involved)
Examples	int, float, char, boolean	String, Array, Class, Object

Example of Primitive Data Types:

```
public class Main {
  public static void main(String[] args) {
    int num = 10;
    double price = 99.99;
    boolean isAvailable = true;
    char grade = 'A';

    System.out.println("Number: " + num);
    System.out.println("Price: " + price);
    System.out.println("Available: " + isAvailable);
    System.out.println("Grade: " + grade);
  }
}
```

Example of Reference Data Types:

```
public class Main {
    public static void main(String[] args) {
        String name = "OpenAI";   // String is a reference type
        int[] arr = {1, 2, 3, 4}; // Array is a reference type

        System.out.println("Name: " + name);
        System.out.println("Array: ");
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

Explain the concept of interfaces and abstract classes with examples.

Answer: Abstract Classes

Java abstract class is a class that can not be instantiated by itself, it needs to be subclassed by another class to use its properties. An abstract class is declared using the "abstract" keyword in its class definition.

```
abstract class Shape
{
    int color;
    // An abstract function
    abstract void draw();
}
```

In Java, the following some important observations about abstract classes are as follows:

1). An instance of an abstract class can not be created.
2). Constructors are allowed.
3). We can have an abstract class without any abstract method.
4). There can be a final method in abstract class but any abstract method in class(abstract class) can not be declared as final or in simpler terms final method can not be abstract itself as it will yield an error: "Illegal combination of modifiers: abstract and final".
5). We can define static methods in an abstract class.
6). We can use the abstract keyword for declaring top-level classes (Outer class) as well as inner classes as abstract

7). If a class contains at least one abstract method then compulsory should declare a class as abstract .

8). If the Child class is unable to provide implementation to all abstract methods of the Parent class then we should declare that Child class as abstract so that the next level Child class should provide implementation to the remaining abstract method.

Example:

```
// Abstract class
abstract class Sunstar {
   abstract void printInfo();
}

// Abstraction performed using extends
class Employee extends Sunstar {
   void printInfo()
   {
      String name = "avinash";
      int age = 21;
      float salary = 222.2F;

      System.out.println(name);
      System.out.println(age);
      System.out.println(salary);
   }
}

// Base class
class Base {
   public static void main(String args[])
   {
      Sunstar s = new Employee();
      s.printInfo();
   }
}
```

Interfaces:

An **Interface in Java** programming language is defined as an abstract type used to specify the behaviour of a class. An interface in Java is a blueprint of a behaviour. A Java interface contains static constants and abstract methods.

1). The interface in Java is *a* mechanism to achieve abstraction.

2). By default, variables in an interface are public, static, and final.

3). It is used to achieve abstraction and multiple inheritances in Java.
4). It is also used to achieve loose coupling.
5). In other words, interfaces primarily define methods that other classes must implement.
6). An interface in Java defines a set of behaviours that a class can implement, usually representing an IS-A relationship, but not always in every scenario.

Example:

```
interface testInterface {

   // public, static and final
   final int a = 10;

   // public and abstract
   void display();
}

// Class implementing interface
class TestClass implements testInterface {

   // Implementing the capabilities of
   // Interface
   public void display(){
     System.out.println("Geek");
   }
}
```

Explore multithreading in Java to perform multiple tasks concurrently.

Answer: Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms :

1). Extending the Thread class
2). Implementing the Runnable Interface

Thread creation by extending the Thread class

We create a class that extends the **java.lang.Thread** class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

Example:

```
class MultithreadingDemo extends Thread {
        public void run()
        {
                try {
                        // Displaying the thread that is running
                        System.out.println(
                                "Thread " + Thread.currentThread().getId()
                                + " is running");
                }
                catch (Exception e) {
                        // Throwing an exception
                        System.out.println("Exception is caught");
                }
        }
}

// Main Class
public class Multithread {
        public static void main(String[] args)
        {
                int n = 8; // Number of threads
                for (int i = 0; i < n; i++) {
                        MultithreadingDemo object
                                = new MultithreadingDemo();
                        object.start();
                }
        }
}
```

Output

```
Thread 15 is running
Thread 14 is running
Thread 16 is running
Thread 12 is running
Thread 11 is running
Thread 13 is running
Thread 18 is running
Thread 17 is running
```

Thread creation by implementing the Runnable Interface

We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

Example:

```
class MultithreadingDemo implements Runnable {
	public void run()
	{
		try {
			// Displaying the thread that is running
			System.out.println(
				"Thread " + Thread.currentThread().getId()
				+ " is running");
		}
		catch (Exception e) {
			// Throwing an exception
			System.out.println("Exception is caught");
		}
	}
}

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

```
                    int n = 8; // Number of threads
                    for (int i = 0; i < n; i++) {
                              Thread object
                                        = new Thread(new MultithreadingDemo());
                              object.start();
                    }
          }
}
```

Output:

```
Thread 13 is running
Thread 11 is running
Thread 12 is running
Thread 15 is running
Thread 14 is running
Thread 18 is running
Thread 17 is running
Thread 16 is running
```