**ASSIGNMENT-3**

**Object Oriented Programming(EC605A)**

**Name: RESHMI GANGULY**

**ROLL NO:ECE2017/106**

1. What is the advantage of using Variable length argument list?

Ans)Using variable length arguments in methods gives the following advantages:

* the program code is simplified, since there is no need to declare many implementations of overloaded methods that take a different number of arguments;
* any number of any arguments of a given type can be passed to the method;
* an arbitrary number of arguments of any type can be passed to the method. In this case, the method should take variable length arguments of type Object. Such passing does not require any additional changes in the program;
* you do not need to first form an array of values to pass it to the method. The argument values are passed to the method, separated by commas.

1. Mention few advantages of ArrayList over Array.

|  |  |
| --- | --- |
| **ArrayList** | **Arrays** |
| a) ArrayList is dynamically grows list if internal array out of space, then its size increases by double | a) Arrays is fixed size list and it cannot be modified  oncecreated. |
| b) Previous to Java version 5 storing primitives type was not allowed only object storage was allowed in ArrayList. After introducing Auto-boxing with Java 5 which allows us to store primitive type which get converted to Objects. | b) On the other hand Arrays can store both Objects and  primitive types. |
| c) ArrayList generics are allowed so that one that one can find what kind of object ArrayList going to store. | c)Use of generics is not allowed in Arrays because  Arrays initialize with its type of object it’s going to  store. |
| d)To get the length you will have to call size method. | d)To get the length you will have to call length. |
| e) There is an overhead related to managing the size of the internal array and more if you try to access its element using casting object. | e) This overhead not there in Arrays because initially  youdefine it. |
| f)ArrayList is dynamically allocated elements. | f) Whereas Arrays is statically allocated elements. |

**The ArrayList has following advantages over array:**

A)We can define ArrayList as **re-sizable array**. Size of the ArrayList is not fixed. ArrayList can grow and shrink dynamically.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | classArrayListDemo  {      publicstaticvoidmain(String[] args)      {          ArrayList<String> list = newArrayList<String>();            list.add("ONE");            list.add("TWO");            list.add("THREE");            System.out.println(list.size());     //Output : 3            //Inserting some more elements          list.add("FOUR");            list.add("FIVE");            System.out.println(list.size());    //Output : 5            //Removing an element          list.remove("TWO");            System.out.println(list.size());    //Output : 4      }  } |

B) Elements can be inserted at or deleted from a particular position.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | classArrayListDemo  {      publicstaticvoidmain(String[] args)      {          ArrayList<String> list = newArrayList<String>();            list.add("ZERO");            list.add("TWO");            list.add("FOUR");            System.out.println(list);     //Output : [ZERO, TWO, FOUR]            list.add(2, "THREE");       //Inserting an element at index 2            list.add(1, "ONE");     //Inserting an element at index 1            System.out.println(list);    //Output : [ZERO, ONE, TWO, THREE, FOUR]            list.remove(3);       //Removing an element from index 3            System.out.println(list);    //Output : [ZERO, ONE, TWO, FOUR]      }  } |

C) ArrayList class has many methods to manipulate the stored objects.

ArrayList class has methods to perform solo modifications ( add(), remove()… ), bulk modifications ( addAll(), removeAll(), retainAll()… ), searching( indexOf(), lasIndexOf() ) and iterations( iterator() ).

1. What are AUTOBOXING and UNBOXING?

# Ans)

*Autoboxing* is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called *unboxing*.

Here is the simplest example of autoboxing:

Character ch = 'a';

Consider the following code:

List<Integer> li = new ArrayList<>();

for (int i = 1; i< 50; i += 2)

li.add(i);

Although you add the int values as primitive types, rather than Integer objects, to li, the code compiles. Because li is a list of Integer objects, not a list of int values, you may wonder why the Java compiler does not issue a compile-time error. The compiler does not generate an error because it creates an Integer object from i and adds the object to li. Thus, the compiler converts the previous code to the following at runtime:

List<Integer> li = new ArrayList<>();

for (int i = 1; i< 50; i += 2)

li.add(Integer.valueOf(i));

Converting a primitive value (an int, for example) into an object of the corresponding wrapper class (Integer) is called autoboxing. The Java compiler applies autoboxing when a primitive value is:

* Passed as a parameter to a method that expects an object of the corresponding wrapper class.
* Assigned to a variable of the corresponding wrapper class.

Consider the following method:

public static int sumEven(List<Integer> li) {

int sum = 0;

for (Integer i: li)

if (i % 2 == 0)

sum += i;

return sum;

}

Because the remainder (%) and unary plus (+=) operators do not apply to Integer objects, you may wonder why the Java compiler compiles the method without issuing any errors. The compiler does not generate an error because it invokes the intValue method to convert an Integer to an int at runtime:

public static int sumEven(List<Integer> li) {

int sum = 0;

for (Integer i : li)

if (i.intValue() % 2 == 0)

sum += i.intValue();

return sum;

}

Converting an object of a wrapper type (Integer) to its corresponding primitive (int) value is called unboxing. The Java compiler applies unboxing when an object of a wrapper class is:

* Passed as a parameter to a method that expects a value of the corresponding primitive type.
* Assigned to a variable of the corresponding primitive type.

The [Unboxing](https://docs.oracle.com/javase/tutorial/java/data/examples/Unboxing.java) example shows how this works:

import java.util.ArrayList;

import java.util.List;

public class Unboxing {

public static void main(String[] args) {

Integer i = new Integer(-8);

// 1. Unboxing through method invocation

int absVal = absoluteValue(i);

System.out.println("absolute value of " + i + " = " + absVal);

List<Double>ld = new ArrayList<>();

ld.add(3.1416); // Π is autoboxed through method invocation.

// 2. Unboxing through assignment

double pi = ld.get(0);

System.out.println("pi = " + pi);

}

public static int absoluteValue(int i) {

return (i< 0) ? -i : i;

}

}

The program prints the following:

absolute value of -8 = 8

pi = 3.1416

Autoboxing and unboxing lets developers write cleaner code, making it easier to read. The following table lists the primitive types and their corresponding wrapper classes, which are used by the Java compiler for autoboxing and unboxing:

|  |  |
| --- | --- |
| **Primitive type** | **Wrapper class** |
| boolean | Boolean |
| byte | Byte |
| char | Character |
| float | Float |
| int | Integer |
| long | Long |
| short | Short |
| double | Double |

1. What is Dynamic Binding? Explain with examples.

### Ans) **Dynamic Binding or Late Binding:**

When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or late Binding. [Method Overriding](https://beginnersbook.com/2014/01/method-overriding-in-java-with-example/) is a perfect example of dynamic binding as in overriding both parent and child classes have same method and in this case the **type of the object** determines which method is to be executed. The type of object is determined at the run time so this is known as dynamic binding.

**Dynamic binding example**

This is the same example that we have seen above. The only difference here is that in this example, overriding is actually happening since these methods are **not** static, private and final. In case of overriding the call to the overriden method is determined at runtime by the type of object thus late binding happens. Lets see an example to understand this:

classHuman{

//Overridden Method

publicvoidwalk()

{

System.out.println("Human walks");

}

}

classDemoextendsHuman{

//Overriding Method

publicvoidwalk(){

System.out.println("Boy walks");

}

publicstaticvoidmain( Stringargs[]) {

/\* Reference is of Human type and object is

\* Boy type

\*/

Humanobj = newDemo();

/\* Reference is of HUman type and object is

\* of Human type.

\*/

Human obj2 = newHuman();

obj.walk();

obj2.walk();

}

}

**Output:**

Boy walks

Human walks

1. What is the difference between overriding a superclass method and overloading a superclass method?

Ans)Overriding and Overloading are two very important concepts in Java. They are confusing for Java novice programmers. This post illustrates their differences by using two simple examples.

1. Definitions:

*Overloading* occurs when two or more methods in one class have the same method name but different parameters.

*Overriding* means having two methods with the same method name and parameters (i.e., *method signature*). One of the methods is in the parent class and the other is in the child class. Overriding allows a child class to provide a specific implementation of a method that is already provided its parent class.

2. Overriding vs. Overloading:

Here are some important facts about Overriding and Overloading:

1). The real object type in the run-time, not the reference variable's type, determines which overridden method is used at *runtime*. In contrast, reference type determines which overloaded method will be used at *compile time*.  
2). Polymorphism applies to overriding, not to overloading.  
3). Overriding is a run-time concept while overloading is a compile-time concept.

3. An Example of Overriding:

Here is an example of overriding. After reading the code, guess the output.

|  |
| --- |
| **class**Dog{  **publicvoid**bark(){  System.out.println("woof ");  }  }  **class** Hound **extends**Dog{  **publicvoid**sniff(){  System.out.println("sniff ");  }    **publicvoid**bark(){  System.out.println("bowl");  }  }    **publicclass**OverridingTest{  **publicstaticvoid**main(String[]args){  Dog dog=**new**Hound();  dog.bark();  }  } |

**Output**:

bowl

In the example above, the dog variable is declared to be a Dog. During compile time, the compiler checks if the Dog class has the bark() method. As long as the Dog class has the bark() method, the code compilers. At run-time, a Hound is created and assigned to dog. The JVM knows that dog is referring to the object of Hound, so it calls the bark() method of Hound. This is called Dynamic Polymorphism.

4. An Example of Overloading:

|  |
| --- |
| **class**Dog{  **publicvoid**bark(){  System.out.println("woof ");  }    *//overloading method*  **publicvoid**bark(**int**num){  **for**(**int**i=0;i<num;i++)  System.out.println("woof ");  }  } |

In this overloading example, the two bark method can be invoked by using different parameters. Compiler know they are different because they have different method signature (method name and method parameter list).

1. What is “Is-a” relationship? How is it different from “Has-a” relationship?

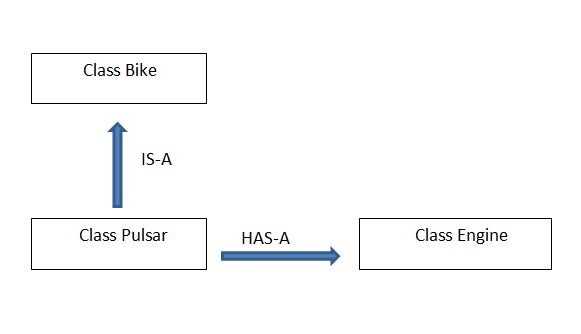
Introduction:

In Java, we can reuse our code using an Is-A relationship or using a Has-A relationship. An Is-A relationship is also known as inheritance and a Has-A relationship is also known as composition in Java.

Is-A Relationship in Java:

In Java, an Is-A relationship depends on inheritance. Further inheritance is of two types, class inheritance and interface inheritance. It is used for code reusability in Java. For example, a Potato is a vegetable, a Bus is a vehicle, a Bulb is an electronic device and so on. One of the properties of inheritance is that inheritance is unidirectional in nature. Like we can say that a house is a building. But not all buildings are houses. We can easily determine an Is-A relationship in Java. When there is an extends or implement keyword in the class declaration in Java, then the specific class is said to be following the Is-A relationship.

Has-A Relationship in Java:

In Java, a Has-A relationship is also known as composition. It is also used for code reusability in Java. In Java, a Has-A relationship simply means that an instance of one class has a reference to an instance of another class or an other instance of the same class. For example, a car has an engine, a dog has a tail and so on. In Java, there is no suchkeyword that implements a Has-A relationship. 

**Example:**

1. **package** relationsdemo;
2. **public** **class** Bike
3. {
4. **private** String color;
5. **private** **int** maxSpeed;
6. **public** **void** bikeInfo()
7. {
8. System.out.println("Bike Color= "+color + " Max Speed= " + maxSpeed);
9. }
10. **public** **void** setColor(String color)
11. {
12. **this**.color = color;
13. }
14. **public** **void** setMaxSpeed(**int** maxSpeed)
15. {
16. **this**.maxSpeed = maxSpeed;
17. }
18. }

In the code above the Bike class has a few instance variables and methods.

1. **package** relationsdemo;
2. **public** **class** Pulsar **extends** Bike
3. {
4. **public** **void** PulsarStartDemo()
5. {
6. Engine PulsarEngine = **new** Engine();
7. PulsarEngine.stop();
8. }
9. }

Pulsar is a type of bike that extends the Bike class that shows that Pulsar is a Bike. Pulsar also uses an Engine's method, stop, using composition. So it shows that a Pulsar has an Engine.

1. **package** relationsdemo;
2. **public** **class** Engine
3. {
4. **public** **void** start()
5. {
6. System.out.println("Started:");
7. }
8. **public** **void** stop()
9. {
10. System.out.println("Stopped:");
11. }
12. }

The Engine class has the two methods start( ) and stop( ) that are used by the Pulsar class.

1. **package** relationsdemo;
2. **public** **class** Demo
3. {
4. **public** **static** **void** main(String[] args)
5. {
6. Pulsar myPulsar = **new** Pulsar();
7. myPulsar.setColor("BLACK");
8. myPulsar.setMaxSpeed(136);
9. myPulsar.bikeInfo();
10. myPulsar.PulsarStartDemo();
11. }
12. }

In the code above we make an object of the Pulsar class and then initialize it. All the methods like setColor( ), bikeInfo( ), setMaxSpeed( ) are used here because of the Is-A relationship of the Pulsar class with the Bike class.

**Output:**

Bike color= Black Max Speed= 136

Stopped**:**