Overloading:

It is when two methods in the same class have the same name, provided that their signature is different.

So what is a method signature?

A method's signature is the method's name, the number of formal parameters it has and the type of each parameter. All of these combined compose a method's signature.

When we have two methods that are overloaded, they either have a different number of parameters or different types of the actual parameters that you provide.

Very Important Note: the method signature does not include the method's return type. It is illegal to have two subroutines in the same class that have the same signature but that have different return types. For example, it would be a syntax error for a class to contain two methods defined

```
int getln()

{

……

}

double getln()

{

}
```

Wrapper Class and Auto-boxing:

There is a class for each of the primitive data types (int, short, long, byte, Boolean, float, double etc.). These classes are called wrapper classes. Although they contain useful static methods, they have another use as well. They are used for creating objects that represent primitive type values.

Recall: that primitive types are not classes, and values of primitive type are not objects, however it is sometimes useful to treat a primitive value as if it were an object. You can't do that literally but you can "wrap" the primitive type value in an object belonging to one of the wrapper classes.

Example:

Let us initialize an object of type Double that contains a single instance variable of type double. The object is said to be a "wrapper" for the double value.

Double d= new Double(5.05);

The value of d in the example contains the same information as the value of type double, but it is an object. If you want to retrieve the double value that is "wrapped" in the object, we have to "unwrap" the object. We can easily do this by using the function .doubleValue();.

In the example above we would write d.doubleValue();

Note: This is also especially useful with collections because they can only hold objects. So if we wanted to add a primitive type value into a collection it has to be put into a wrapper object first).

Please, also note that this example is also applicable to all the other primitive data types as well.

Let's say that we wanted to use the new primitive object with mathematical manipulations. So taking the example (d) above I want to multiply it by two. In the old java versions I would have had to "unwrap" the primitive object first and then I would've been able to use it in mathematical manipulations.

Now in the new java versions, this isn't necessary due to a feature called auto-boxing. If we wanted to multiply (d) by two we would simply write d*2. The compiler would automatically unbox the object (d) and multiply it by 2.

This also works the other way around if we wanted to use a value of type int in a context that requires an object of type Integer, the int will automatically be wrapped in an Integer object.

Example

If I wrote this line of code: -

Integer answer=42;

The compiler will read this as if it were

Integer answer= new Integer(42);

The Class “Object”:

A very notable feature in object oriented programming is the ability to create subclasses of a class. This subclass inherits all the properties or behaviors of the class but can modify what it inherits. This is known as inheritance and will be found later on in my notes.

However, what is really breathtaking is when you create a class in java and don’t explicitly make it a sub-class of some other class, then it automatically becomes a sub-class of a special class named **Object.**

So what is the class **Object**?

The class **Object** is the one class that is not a sub-class of any other class in java. This class defines several methods that are inherited by every other class. These methods can be used with any object whatsoever. One example of the is the method .toString(); which returns a value of type String that is supposed to be a string representation of the object.

	Strings
Definition	-) They are basically an array of characters. The array of characters can form words or whatever the user or programmer wants. -) Every single character is defined in the ASCII table and thus we can combine any ASCII character to create a string.
Advantages	We can use strings to write coherent sentences in any language in the world. Strings form one of the primitive data types in any coding language.
Disadvantages	N.A
Imported Library	N.A already exists within standard java library.
Declaration	private String s; private String s= “adnan”;
Constructor Implementation/Initialization	N.A

Common Built-in Functions and their uses	.equals(); //checks if the string identity is equal. Always use this function when checking if two strings are the same.
	.substring();
	public String substring(int beginIndex)
	Returns a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string.
	
	Examples:
	
	"unhappy".substring(2) returns "happy"
	"Harbison".substring(3) returns "bison"
	"emptiness".substring(9) returns "" (an empty string)

Method Implementation	Checking whether a string is equal to another or not.
	String input= “hello”;
	if(input.equals("bye"))
	{
	System.out.println(input);
	}

	Arrays
Definition	-) An array is a container that holds a fixed number of values of the same type. -) Each item in the array is an element. -) Each element is accessed by a numerical index.
Advantages	-) It adds/removes elements of one type into a matrix. -) the matrix can be searched and sorted using several methods that the programmer can code.
Disadvantages	-) Only elements of the same type can be added. -) Duplication of data is a huge problem with arrays. -) They are static in nature and have a fixed size which cannot be altered after decleration.
Imported Library	It's already included with the standard java implementation
Declaration	-) private PrimitivedataType arrayName []; *** array of objects -) private Classname arrayname []; **This declaration means that the elements of the array are variables belonging to the specific class of this array. Like any object variable <u>each element of the array can either be null or can hold a reference to an object.</u>** **Please note that the term array of objects can be misleading, since the objects are not in the array. The array can only contain references to objects.** **Multi-dimensional arrays -) private PrimitivedataType arrayName[][];
Constructor Implementation/Initialization	-) primtiveDatatype Arrayname[]= new Dataype[ArraySize]; *** Array of Objects -) Classname Arrayname[]= new Classname[ArraySize]; *** Multi-dimensional arrays PrimitivedataType arrayname[][]= new Datatype[row size][column size];

Common Built-in Functions and their uses	Arrayname.length (This returns the size of the array) This function is very useful when implementing arrays in for-loops. This function can also be manipulated in for loops to create an iteration that is smaller than the total number of elements in the array. Ex) For(int i=0; i<array.length-1;i++) // this will loop all the elements in the array except for the last one. We can also do other manipulation within the square brackets[]. Ex) 1) array[i-1]=array[i]; 2) array[array.length-1]=c;

Method Implementation	Printing array elements (primitive array)
	** assume array is of type integer.
	
	for(int i=0; i<array.length;i++)
	{
	System.out.printf("%i",array[i]);
	}
	
	Printing array elements
	(array of objects)
	** assume class Student with array name (students)
	
	For(int i=0; i<students.length;i++)
	{
	System.out.printf("Student:%s\nGrade=%i\n",students[i].getname(),students[i].getgrade());
	}
	
	**Why %s and %i?
	Because student.getname() returns a value of type string.
	
	While student.getgrade() returns a value of type integer.

Printing Multi-dimensional array elements(primitive array)

```
for(int i=0; i<size(row);i++)
{
For(int j=0; j<size(column);j++)
{
System.out.printf("[%d][%d]=%s",i,j,array[i][j]);
}
}
```

**assume array is of type string.

	ArrayLists
Definition	ArrayLists are a programming implementation that go under the branch of collections. So in order to define ArrayLists we are going to define collections. -) So collections are objects which group multiple elements into a single unit. -) The ArrayList class is an implementation of a collection. -) collections have numerous methods for object management.
Advantages	-) A collection is dynamic; it has no fixed size. It grows or becomes smaller depending if you add or remove elements from it. -) Thus they are dynamic in nature. -) They have object representations for the primitive data types. -) it works really well with for-loops and for-each loops.
Disadvantages	N.A
Imported Library	Import java.util.ArrayLists;

Declaration	One Block Declaration and Initialization
	
	ArrayList<Class Name> ArraylistName= new ArrayList<>();
	
	***With Generics
	
	Private ArrayList<Class Name> ArraylistName;
	
	*** Without Generics
	
	Private ArrayList<> ArraylistName;

Constructor Implementation/Initialization	***With Generics
	ArraylistName= new ArrayList<Class Name>();
	
	***Without
	ArrayListName= new ArrayList<>();

Common Built-in Functions and their uses	.size(); // returns the number of elements.
	.add(); // adds an element to the collection.
	.remove();// removes an element from the collection.
	.clear(); //removes all elements from the collection.
	.contains();// checks whether the specified object passed as parameter exists in the collection or not.
	.get(); //returns an object at a specified index. (if we want a specific attribute of all the objects in the ArrayList we must equate it to a variable of the same class so that we can use that class's methods (for-each/ for-loop) or indicate an index in this method).
	isEmpty();// returns true if list has no elements.

Method Implementation

(With Generics Code)

q) Create a java code that sets and gets an ArrayList.(Assume the ArrayList is declared and initialized called Names).

```
Public void setNames(ArrayList<String> xnames)
{
Names=xnames;
}

Public ArrayList<String> getNames()
{
return Names;
}
```

***All the ArrayLists code below is created with an assumption that they were declared with generics.

Printing from an ArrayList

```
for(int i=0; i<ArraylistName.size();i++)
{
System.out.println(ArraylistName.get(i));
}
```

Using the .add() ArrayList method to add an entire object Student that has constructor parameters of (String n, double g).

```
Math101.add(new Student("Bob",2.9)); //ArrayList name is math101.
```

Checking whether an object exists within the ArrayList (assume that the following code checks whether a bank account exists or not).

```
public Account getAccount(int n)
   {
      if(accounts.contains(n))
     {
        return accounts.get(n);
   }
   else
   return null;
   }
//notice that with generics we can chain the above to get specific
object attributes.
```

Using a for-each loop to print objects of an ArrayList that are declared in the class (client) but the ArrayList is of type Account.

```
public void printClerkPhoneNumber()
    {
        for(Account s: accounts)
        {
        if (s.getClerk()!= null)
            System.out.println(s.getClerk().getName() + " has phone
number " + s.getClerk().getPhoneNb());
        else
            System.out.println("Sorry, there is no clerk assigned to your
account!");
        }
    }
```

Using the ArrayList along with other methods to manipulate the objects within the ArrayList. (The ArrayList is called accounts and is of type Account). This entire method was created in a different class.

```
    public void TakeOffMoney(float sum, int n)
    {
        if(accounts.get(n)!=null){
        accounts.get(n).TakeOff(sum);
       printBalance(); //method within class
    }
    else
    {
        accounts.get(0).TakeOff(sum);
        printBalance();
    }
    }
```

Important Question: Find the factors of a number while using generics. Please note that the ArrayList is of type Integer (Class Integer).

```
import java.util.*;
public class Factors
{
   private int number;
   private ArrayList<Integer> factors; //generics decleration.

   public Factors(int x)
   {
      if(x<=0)
      {
         number=1;

      }
      else
      {
         number=x;
      }
      factors= new ArrayList<Integer>(); //ArrayList initialization.
      for(int i=1;i<=x;i++)
      {
         if(x%i==0)
         {
            factors.add(i); //adding the values into the ArrayList.
         }

      }
   }

   public void printnumber()
   {
      System.out.println(number);
      int prod=1;
      for(int l: factors)
      {
         prod*=l;
         System.out.println(l); //printing the factors of the number.

      }
      System.out.println(prod); //printing the products of the factors.
   }

}
```

(Without Generics Code)
***All the ArrayLists code below is created with an assumption that they were declared without generics.

Printing the contents of an ArrayList while also calling other methods.

```
        for(int i=0;i<accounts.size();i++)
        {
            Account c= (Account) accounts.get(i); //we have to cast
            System.out.println("The balance is" + c.getBalance());
            System.out.println("The number is" + c.getnumber());
            System.out.println("The credit is" + c.getCredit());
        }
    }
```

Checking whether an object exists within the ArrayList (assume that the following code checks whether a bank account exists or not).

```
public Account getAccount(int n)
    {
           if(accounts.contains(n))
        {
            return (Account)accounts.get(n); //notice the casting
    }
    else
    return null;
    }
}
```

Important Question: Find the factors of a number without using generics that are of type Integer (Class Integer).

```
import java.util.*;
public class Factors2
{
   private int number;
   private ArrayList factors; //declaration without generics.

   public Factors2(int x)
   {
      if(x<=0)
      {
         number=1;

      }
      else
      {
         number=x;
      }
      factors= new ArrayList<>(); //initialization without generics.
      for(int i=1;i<=x;i++)
      {
         if(x%i==0)
         {
            factors.add(new Integer(i)); //Since we are adding integer
objects we must add them as if we are adding objects.
// new Integer(i) has a constructor that takes an integer number.
         }

      }
   }

   public void printnumber()
   {
      System.out.println(number); //prints the number we're working
on.
      int prod=1;
      Iterator itr= factors.iterator();
      while(itr.hasNext())
      {
         Integer x= (Integer)itr.next(); //we have to cast (this is crucial).
         prod=prod*x.intValue(); //intValue converts Integer to int.
         System.out.println(x.intValue());
      }
      System.out.println(prod);
   }

}
```

Adding entire objects to the ArrayList with generics is the same without generics.

Math101.add(new Student(“Bob”,2.9)); //ArrayList name is math101.

Adding/removing/ checking an ArrayList declared without generics is done normally for an entire object. However manipulating specific object attributes requires using casting and declaring a new object of the same class that the ArrayList belongs to.

*****(Method Chaining) (This is a general rule and isn’t just applicable for ArrayLists).

We are going to assume that we have two classes Client and Account.

1) If we want both classes to see each other, then a field of type Client should be created in the class Account and vice versa.

2) When for example we want to call the methods in the other class we can only do so through the field that is of the same type of that class.

   Ex) Assume that we have the field (private Account accounts) in the class client.

   So we can start chaining methods like so

   EX2) accounts.getBalance(); **getBalance is a method in the class Account.**

3) If the method has a return type of either void or any primitive data type, then you will only be able to chain one method like in the example above.

	4) However, if you were calling a method that returns an entire object, then you would be actually able to chain other methods of that object. (A field of that class must also be declared, look at note 1). EX3) Assume that we have a third class called clerk. So we will do this… accounts.getClerk().getName(); The method getClerk() has a return type of class Clerk. So this entire method returns an entire object of type Clerk. But for example we don't want the entire object we want specific things about that object. So we chain it with other methods that are in the class clerk to get what we want. (please note that the chaining in this example is only possible with methods from the class clerk).

	Iterator
Definition	-) Iterators are objects that allow you to traverse the elements of a collection in sequence. -) Collections have an iterator method that allows you to traverse their elements through an iterator.
Advantages	-) They allow you to traverse a collection with ease. -) They work with all different types of collections. Including and not limited to (ArrayLists and Sets).
Disadvantages	N.A
Imported Library	Import java.util.Iterator();
Declaration	It is usually declared in one sentence through a specific method that requires the use of the iterator. -) Iterator Iteratorname= CollectionName.iterator();
Constructor Implementation/Initialization	It can be done but I've never seen any question that requires it. You need to remember that the iterator itself isn't a collection but an object that allows you to make efficient use of the collections that you are dealing with.

Common Built-in Functions and their uses	.hasNext(); // checks if there is a next element in the collection.
	.next(); //retrieves the next element in the collection, if it is present.
	.remove(); // removes the element that the pointer is pointing at in the collection.
	.hasPrevious(); //true if there is a previous element. obj
	.previous(); //Returns the previous element.
	.nextIndex();// Returns index of element that would be returned by subsequent call to next().
	.previousIndex();// Returns index of element that would be returned by subsequent call to previous().

Method Implementation	**** Let us first assume that the collections were declared with generics. **** Secondly, note that iterators are most commonly used with while statements and sometimes they are used with for loops.

Ex) Printing elements from a collection using an iterator.
Assume the collection is of type Account and the collection itself is called accounts.

```
public void printBalance() //function name
   {
       Iterator<Account> itr= accounts.iterator();
       while(itr.hasNext())
       {
       Account c= itr.next();
       System.out.println("Balance is " +c.getBalance() );
       System.out.println("account number is " + c.getnumber());

   }
   }
```

Using the iterator to print elements in a collection.
Assume the following declarations
ArrayList<String> alist = new ArrayList<String>();

```
for (Iterator<String> it = alist.iterator(); it.hasNext(); )
{
//notice that within the for-statement we declared the iterator.
String s = it.next();
System.out.println(s);
}
```

The previous for statement can be substituted by for each loop as follows:

```
for (String s : alist) {
System.out.println(s);
}
```

Removing a specific element from a collection. Assume the following declaration.
Iterator<Student> it = students.iterator();

```
while(it.hasNext())
{
Student s = it.next();
String drop = s.getname();
if(drop.equals(nameToRemove)) //.equals is a string method that is explained in strings notes.
 {
it.remove();
}
}
```

	**** Without generics
	Printing using an iterator
	`public void printallaccountsiterator()`
	`{`
	`    Iterator itr= accounts.iterator();`
	`    while(itr.hasNext())`
	`    {`
	`        Account c= (Account) itr.next(); //notice casting.`
	`        System.out.println("The balance is" + c.getBalance());`
	`        System.out.println("The number is" + c.getnumber());`
	`        System.out.println("The credit is" + c.getCredit());`
	`    }`
	`}`
	
	******* anything else such as removing or checking what’s next using an iterator is the same with or without generics. Unless you want to check specific element attributes we will need to use casting as in the example above.

	Sets
Definition	-) Sets are actually an interface in java that allows for storing elements in an unordered manner. But ensures that there are no duplicate elements in one set. -) Since Sets are an interface, they include two collections/implementations that we will be using. The first implementation is known as the Hashset and the other is called TreeSet.
Advantages	-) Sets are exactly implemented like collections so they share the same built in functions like ArrayLists. -) Sets can also be used with iterators as they also do have an iterator function like ArrayLists. ***** Difference between Hashset, TreeSet and ArrayList. -) Hashset: A collection of elements that contains no duplicate elements -) TreeSet: A sorted collection of elements that contains no duplicate elements -) ArrayList: A dynamic array implementation
Disadvantages	N.A
Imported Library	Import java.util.HashSets; But instead just use import java.util.*; (this statement imports everything we need).

Declaration	***** With Generics
	private HashSet<Class Name> Setname;
	private TreeSet<Class Name> Setname;
	
	***** Without Generics
	private HashSet<> Setname;
	private TreeSet<> Setname;
	
	***** One block Declaration and initialization
	
	HashSet<Class Name> Setname = new HashSet< >();
	TreeSet<Class Name> Setname= new TreeSet<>();

Constructor Implementation/Initialization	-) Setname= new Hashset<>(); -) Setname= new Treeset<>();

Common Built-in Functions and their uses	.size(); // returns the number of elements.
	.add(); // adds an element to the collection.
	.remove();// removes an element from the collection.
	.clear(); //removes all elements from the collection.
	.contains();// checks whether the specified object passed as parameter exists in the collection or not.

.get(); //returns an object at a specified index. (if we want a specific attribute of all the objects in the ArrayList we must equate it to a variable of the same class so that we can use that class's methods (for-each/ for-loop) or indicate an index in this method).
isEmpty();// returns true if list has no elements.
****** Set interface functions

- s1.containsAll(s2) — returns true if s2 is a subset of s1. (s2 is a subset of s1 if set s1 contains all of the elements in s2.)
- s1.addAll(s2) — transforms s1 into the union of s1 and s2. (The union of two sets is the set containing all of the elements contained in either set.)
- s1.retainAll(s2) — transforms s1 into the intersection of s1 and s2. (The intersection of two sets is the set containing only the elements common to both sets.)
- s1.removeAll(s2) — transforms s1 into the (asymmetric) set difference of s1 and s2. (For example, the set difference of s1 minus s2 is the set containing all of the elements found in s1 but not in s2.)

Method Implementation

***** Implementing sets is exactly like implementing ArrayLists. Consider that we have declared a set called books of type Book and we want to print its contents using an iterator. (With generics declaration).

```
public void printbooks()
   {
       Iterator <Book> itr= books.iterator();
       while(itr.hasNext())
       {
          itr.next();
         itr.next().print();
       }
   }
// notice the code is the same.
```

****** adding and printing to a set using a for each loop.

```
import java.util.HashSet;
...
HashSet<String> mySet = new HashSet<String>();
mySet.add("C Language");
mySet.add("Java Language");
mySet.add("Lisp Language");
for(String language : mySet)
{
//do something with language
}
// once again notice is the same as using ArrayLists.
```

	Maps
Definition	-) They are collections that contain pairs of values. -) pairs consist of a key and a value. -) lookup works by supplying a key, and from that key we get its corresponding value. -) Thus a map is an object that maps keys to values. -) A map cannot contain duplicate keys. -) Each key can map only one value. -) Maps are similar to sets in which they are implemented through an interface, thus we have HashMaps and TreeMaps.
Advantages	-) Duplication cannot occur within data. -) It's highly organized.
Disadvantages	N.A
Imported Library	Import java.util.*; (easier)
Declaration	****With Generics private HashMap<KeydataType, valueDatatype/class> mapName; **** Without Generics private HashMap<> mapName; ****One- line declaration and initialization private HashMap<KeydataType, valueDatatype/class> mapName = new HashMap<>();
Constructor Implementation/Initialization	mapName= new HashMap<>();
Common Built-in Functions and their uses	-) .get(Object key): Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key. -) .isEmpty(): Returns true if this map contains no key-value mappings. -).keySet(): Returns a Set view of the keys contained in this map. -).put(K key, V value): Associates the specified value with the specified key in this map (optional operation). -) .remove(Object key): Removes the mapping for a key from this map if it is present (optional operation). -).size(): Returns the number of key-value mappings in this map. -) .putAll(Map<? extends K,? extends V> m): Copies all of the mappings from the specified map to this map (optional operation).

.entrySet():-
Set<Map.Entry<K,V>> entrySet()
Returns a Set view of the mappings contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator's own remove operation, or through the setValue operation on a map entry returned by the iterator) the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via
the Iterator.remove, Set.remove, removeAll, retainAll and clear operations. It does not support the add or addAll operations.

Returns:

a set view of the mappings contained in this map

*******This function is specifically used to iterate through a map using an iterator.

******* This function also has sub-functions in it that are really important to know.

******* The difference between this function and keySet() is that this function takes the entire map along with its values and not just the keys unlike the keyset() function.

These are the functions for .entrySet();

Modifier and Type	Method and Description
`boolean`	`equals(Object o)` Compares the specified object with this entry for equality.
`K`	`getKey()` Returns the key corresponding to this entry.
`V`	`getValue()` Returns the value corresponding to this entry.
`int`	`hashCode()` Returns the hash code value for this map entry.
`V`	`setValue(V value)` Replaces the value corresponding to this entry with the specified value (optional operation).

Method Implementation	****** It’s very important to know that maps don’t behave like anything we studied before. In order to iterate/traverse through a map we have to copy the map’s keys to a set and from that set we can access the values of the map. It sounds complicated but it’s actually quite easy.
	
	Ex1) Declaring a HashMap and then adding some elements to it. After that we want to print a specific element in that HashMap.
	
	HashMap <String, String> phoneBook = new HashMap<String, String>();
	...
	phoneBook.put("Charles Nguyen", "(531) 9392 4587");
	phoneBook.put("Lisa Jones", "(402) 4536 4674");
	phoneBook.put("William H. Smith", "(998) 5488 0123");
	String phoneNumber = phoneBook.get("Lisa Jones");
	System.out.println(phoneNumber);
	
	Ex2) Printing all the elements in a map using a for-each loop. The map we are implementing in this case is specifically a map containing books of type Book. We want to print the books along with their registration number. (This was done in the class called Library).
	
	Note:- this is the declaration for the following map
	private HashMap<String, Book> registration;
	……..
	public void printlist()
	   {
	       Set<String> key=registration.keySet(); //notice here we copied the keys to a set called key, and we use it in the for-each loop.
	We used the .keySet() function to do so.
	       for(String s:key)
	       {
	          System.out.println(s);
	          registration.get(s).print();// (print is a method we created in another class called Book. registration.get(s) will get us the book at (s) so we can then chain it with the method called print in the book class since registration.get(s) returns an object of type book).
	More so, the print function simply prints the registration number.
	       }
	   }

Ex3) If we want to use an iterator through the map, things become a bit more complicated. Notice that in the above example we used a for-each loop so that case was different. In this example we are going to print the keys with their corresponding values.

import java.util.*;
public class Trmap
{
public static void main(String args[])
{
TreeMap<Integer, String> treemp = new TreeMap<Integer, String>();
//one-line declaration.
treemp.put(1, "One");
treemp.put(20, "Twenty");
treemp.put(50, "Fifty");
treemp.put(30, "Thirty");
treemp.put(7, "Seven");
***** The below code is the important part.
Set set = treemp.entrySet();// we declared a set and equated using the entrySet() function.
Iterator<Map.Entry> itr = set.iterator();**//common set method**
//the (Map.Entry) within the triangular brackets is crucial since the iterator “itr” should know that it must iterate through the entire set which we declared that includes the entire map.
while(itr.hasNext()) **//common iterator method**
{
Map.Entry mapmember = itr.next();
//note, that Map.Entry is an object so we can classify variables based on it and use that variable within the while loop of the iterator.
System.out.print("key: "+ mapmember.getKey() + " , Value: ");
System.out.println(mapmember.getValue());
//.getKey() will retrieve the key and is a function within the entry set, While the getValue() is also a function within the entry set and so will also return the value of that key within the specific iteration.
}
}
}

	Ex4) A similar example to the one above but instead of using TreeMaps we will be using HashMaps. We will also be adding and removing some elements into the HashMap and we will then re-print it to show its results after the modifications.

```
import java.util.*;
public class Hmap
{
public static void main(String args[])
{
HashMap<Integer, String> hashmp = new HashMap<Integer, String>();
hashmp.put(32, "CS115");
hashmp.put(5, "CS331");
hashmp.put(9, "CS470");
hashmp.put(41, "CS419");
hashmp.put(3, "CS212");
//adding initial entries to HashMap.
Set set = hashmp.entrySet();
Iterator<Map.Entry> itr = set.iterator();
while(itr.hasNext())
{
Map.Entry itemx = itr.next();
System.out.print("key: "+ itemx.getKey() + ",Value: ");
System.out.println(itemx.getValue());
}
String val= hashmp.get(3);
/* Get values based on key */
System.out.println("Value of key 3: "+ val);
hashmp.remove(5);
/* Remove values based on key*/
System.out.println("keys and values after removing the item of key
5:");
Set set2 = hashmp.entrySet();
Iterator<Map.Entry> itrx = set2.iterator();
while(itrx.hasNext())
{
Map.Entry maps2 = itrx.next();
System.out.print("Key: "+ maps2.getKey() + " , Value: ");
System.out.println(maps2.getValue());
}
}
}
```

	Inheritance and Polymorphism
Definition	Inheritance is where a "superclass" is created so that it contains shared methods and attributes with other sub-classes. These sub-classes would therefore inherit the methods from the superclass so that they can be called upon when needed. Polymorphism:
Advantages	The main advantage is that code duplication is no longer an issue. The shared code between the sub classes is written only once thus the same code is re-used over and over again. Another advantage is with debugging our code. We only need to check the inherited code once for any errors that might pop up.
Disadvantages	------
Imported Library	------

Declaration	For any sub class we should write this in the very beginning: -
	
	Public class ClassName extends NameofParentClass
	{
	…..
	}
	
	Note: every class has one parent, and that parent will have another parent and so on. So the class that is on the lowest level would in theory inherit everything from its parent class and the other hierarchy of parents linked to it. This of course would need some serious coding skills.

Constructor Implementation/Initialization	Firstly, it must be noted that several classes can be declared as subclasses of the same superclass.
	
	The constructor for any sub class is the same however there are some additions.
	
	1) We should include all the parameters that were passed through the constructor of all parent classes in the constructor of the sub class as well.
	
	public className(Parent1parameter,Parent2parameter…,Subclassparamter)
	{
	Super(parent1parameter,parent2parameter…);
	……..
	}

Common Built-in Functions and their uses	Super(); Used in constructor as seen above and is used to call any parent method.

For example, consider we have a method in the parent class called print();.

If we want to call this method from the child class, then we do this super.print();

When we create an object in subclass, it has the ability to call upon methods within its own class and methods that exist in its parent class.

So if we have a super class called vehicles with its own method and a sub class called cars with its methods as well, an object created within the subclass can use its class methods and its superclass methods.

Now obviously, cars are vehicles so an object of type car is automatically an object of type vehicle. So this brings us to an important fact that

A variable that can hold a reference to an object of class A can also hold a reference to an object belonging to any subclass of A.

Suppose that we have declared the following in the subclass car: -

Car mycar= new Car();

This brings us to the fact that we can declare and initialize objects in several manners: -

Vehicle myvehicle= mycar;
Or
Vehicle myvehicle= new Car();

Either of these statements works, the variable myvehicle holds a reference to a vehicle object that happens to be an instance of the subclass car. The object "remembers" that it is in fact a Car, and not just a vehicle. Information about the actual class of an object is stored as part of that object. It is even possible to test whether a given object belongs to a given class, using the instanceof operator.

Ex)
```
If(myvehicle instanceof Car)
{
….
}
```

On the other hand we can't write

```
Mycar=myvehicle;
```

We can't do this because myvehicle could potentially refer to other types of vehicles (subclasses) that is linked with that aren't cars.
If we wanted to do fix the error in the above code, we would have to use casting.

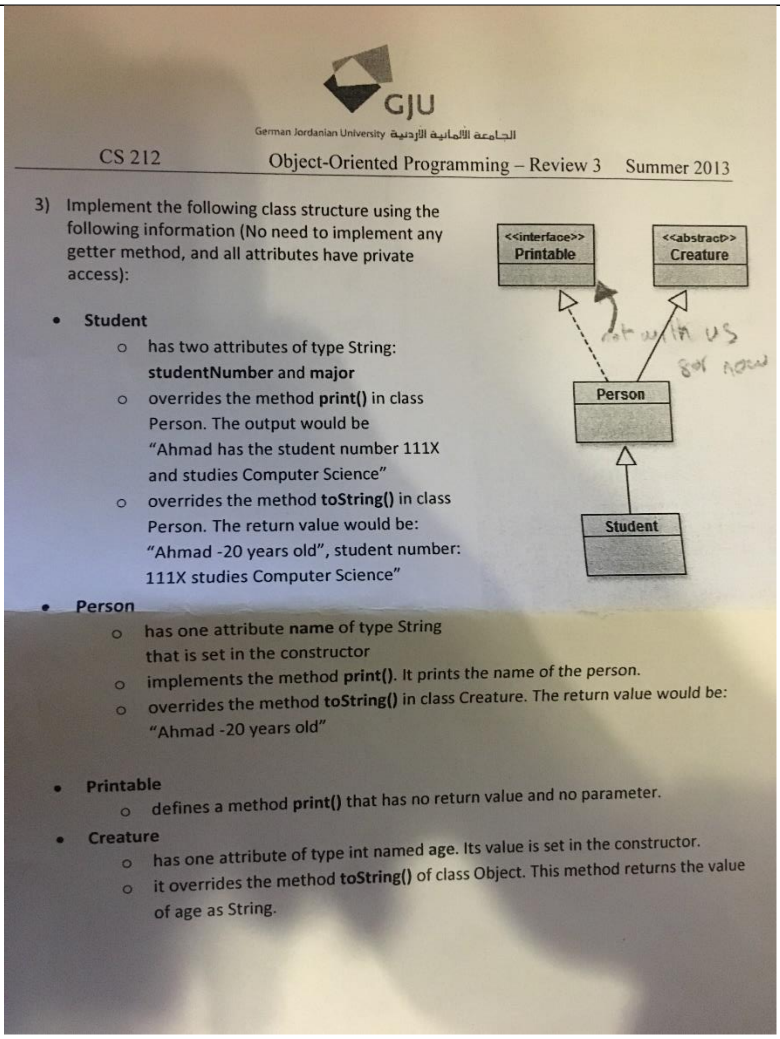
Example)

```
Mycar= (Car)myvehicle;
```

Example 2: -
In this example we are going to look at a practical example involving casting.

```
If(myvehicle instanceof Car)
{
System.out.println("Type of Vehicle: Car.");
Car c;
c=(Car)myvehicle;
//Type cast to get access to numberofdoors method in super class vehicles.
System.out.println("Number of doors: " + c.numberofdorrs);
}
```

Method Implementation



GJU

German Jordanian University الجامعة الألمانية الأردنية

CS 212 Object-Oriented Programming – Review 3 Summer 2013

3) Implement the following class structure using the following information (No need to implement any getter method, and all attributes have private access):

- **Student**
  - has two attributes of type String: **studentNumber** and **major**
  - overrides the method **print()** in class Person. The output would be "Ahmad has the student number 111X and studies Computer Science"
  - overrides the method **toString()** in class Person. The return value would be: "Ahmad -20 years old", student number: 111X studies Computer Science"
- **Person**
  - has one attribute **name** of type String that is set in the constructor
  - implements the method **print()**. It prints the name of the person.
  - overrides the method **toString()** in class Creature. The return value would be: "Ahmad -20 years old"
- **Printable**
  - defines a method **print()** that has no return value and no parameter.
- **Creature**
  - has one attribute of type int named age. Its value is set in the constructor.
  - it overrides the method **toString()** of class Object. This method returns the value of age as String.

1)

```
public class creature
{
private int age;
public creature(int xage)
{
age=xage;
}
public String toString()
{
return “”+ age; // quotations followed by an addition sign and an integer transform the integer to a string.
}
}
```

2)

```
public class person extends creature
{
private String name;
public person(int xage, String xname)
{
super(xage);
name=xname;
}
public void printname()
{
System.out.println(name);
}
public String toString() //method overriding requires using the same method name
{
return name+ super.toString();
}
}
```

3)

```
public class Student extends person
{
private String stno;
private String major;
public Student(int xage,String xname, String xstno, String xmajor)
{
super(xage,xname);
stno=xstno;
major=xmajor;
}
public void printname()
{
super.printname();
System.out.println(stno+ “” + major);
}
Public String toString()
{
Return super.toString;
}
}
```

	Interfaces and Abstract Classes

Definition	Interfaces is Java's attempt at multiple inheritance. Unlike in C++, java doesn't allow multiple inheritance. So the creators of Java created what is called an interface.
	
	***** in an interface, you only write down the methods name with its return type and public/private identity.
	
	***** An interface extends the other.
	
	A class can implement another class just like it can extend another class. The keyword to do so is implements.
	
	A class can implement as many classes as it needs but it can only extend one parent class.
	
	Abstract class: An abstract class is one that is not used to construct objects, but only as a basis for making subclasses. An abstract class exists only to express the common properties of all its subclasses. A class that is not abstract is said to be concrete. You can create objects belonging to a concrete class, but not to an abstract class. A variable whose type is given by an abstract class can only refer to objects that belong to concrete subclasses of the abstract class.
	
	Once you have declared the class to be abstract, it becomes illegal to try to create actual objects in it.
	
	Abstract Methods: An *abstract method* is a method that is declared without an implementation.
	
	****Declaring a **method abstract** means that **method will** be defined in the subclass.
	
	**** The method is defined in the first subclass that extends the abstract class.
	
	**** Any class that then extends the sub-class will have access to that abstract method and so on.
	
	**** If a class is both extending a class and implementing one or more interfaces then we proceed with the extends the implements keywords.
	Ex)
	public example extends example1 implements m1,m2
	{
	……..
	}

Advantages	Same as Inheritance basically.
Disadvantages	N.A
Imported Library	N.A

Declaration	Declaring an interface:
	public interface interfacename
	{
	……..
	}
	Declaring an Abstract class:
	public abstract class classname
	{
	………
	}
	
	Declaring an abstract method
	public abstract datatype methodname(); //recall they have no method bodies, they are initialized in the sub class.
	
	Initializing a void method declared in an interface within an abstract class that implements the interface.
	
	public void methodname(){};
	
	// This method should be initialized in the subclass or they can also be initialized within the abstract class because they have no return type.
	
	If a method has a return type it should be written in the abstract class or in another class that extends the abstract class.
	
	Also please note that when a method is initialized it is now accessible to all of its subclasses. This is very important as is it applicable everywhere.

	GUI (graphical user interface) and Image Processing
Definition	Allows the user to see all the visual elements that make the computer work. (icons, files, images)….
Advantages	It doesn't require the use of the command prompt to use the primary functions for the computer. It becomes user friendly.
Disadvantages	GUI's are memory hungry, therefore faster and larger memories are needed to sustain the heavy load of GUI's.

Imported Library	import javax.swing.*;
	import java.awt.*;
	import java.awt.event.*;
	import java.awt.Color; //image processing

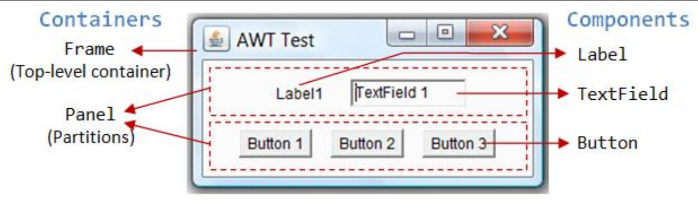
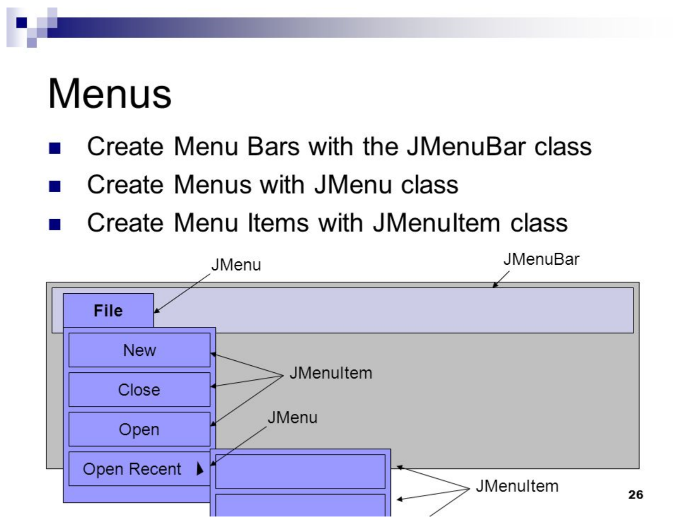
Declaration	private JLabel firstname;
	
	private JButton buttonname;
	
	private JMenuBar menubarname;
	
	private JMenu menuname;
	
	private JMenuItem menuitemname;
	
	private JFrame framename;
	
	private JPanel panelname;
	
	private JTextField textfieldname;
	
	private ImagePanel imagepanelname //image processing

Implementation/Initialization	JFrame framename= new JFrame("frame title");
	JPanel panelname= new JPanel();
	JLabel labelname= new JLabel("label title");
	JTextField textfield name= new JTextField(); //you can leave it empty or add a default text.
	JButton buttonname= new JButton("button title");
	JMenuBar menubarname= new JMenuBar();
	JMenu menuname = new JMenu("menu title");
	JMenuItem menuitemname= new JMenuItem("menu item name");
	Imagepanel= new ImagePanel();

Common Built-in Functions and their uses	Nameofframe.setSize(width,length); // sets the size of the frame.
	
	Panelname.setLayout(new GridLayout(width, length));
	// panels within the frame are specified a certain grid in which they can be put in.

	Panelname.add(GUI element); // adds a GUI element to a panel.
	
	framename.getContentPane().add(panelname);//adds the primary panel to the frame.
	framename.setVisible(true); //this function allows the frame to appear on the screen. This function can also be used with other GUI elements.
	
	frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//puts the close button on the GUI element and allows it to function.
	
	Menubarname.add(GUI item);// adds a GUI element to the menu-bar.
	
	JOptionPane.showMessageDialog(null/framename, "string");// we can concatenate info here.
	
	guielement.isVisible();//Boolean that checks whether a GUI element appears or not.
	
	Gui.setVisible();// Boolean that changes whether a GUI element can be seen or not.
	
	ae.getSource().equals(GUI element)) //checks whether a specific button or menu item is being clicked.
	
	.setText();// sets a specific text. (Can be used for buttons, labels and text fields).
	
	.getText(); // returns a specific text. ((Can be used for buttons, labels and text fields).
	
	contentPane.add(imagepanelname);//image processing
	
	framename.pack();// used with image processing, packs everything to the minimum size requirements.
	
	Imagename.setImage()// sets the image displayed by this icon.
	
	Imagename.lighter(); //lightens the image.
	
	Imagename.threshold(); // simplest function for image segmentation.
	
	Frame.repaint(); will re-color the entire frame if new conditions are applied.

	showStatus(“text goes here”); method that displays information in the form of an applet.
	
	Imagename.getWidth();// returns the width (integer).
	
	Imagename.getHeight();// returns the height (integer).
	
	getpixel(width,height); // sets the coordinates of the pixel (2d).
	
	setPixel(width,height,pixelname);

Method Implementation	 Containers Frame (Top-level container) Panel (Partitions) AWT Test Label1 TextField 1 Button 1 Button 2 Button 3 Components Label TextField Button Menus ■ Create Menu Bars with the JMenuBar class ■ Create Menus with JMenu class ■ Create Menu Items with JMenuItem class JMenu JMenuBar File New Close Open Open Recent JMenuItem JMenu JMenuItem 26 Programming GUI’s is actually quite simple, however it can be a bit tricky. So here are some points to keep in mind at all times. (This is in order). 1) The frame is the primary element in GUI. Everything fits in it, including the primary panel which is added to it. 2) After initializing the frame, it would be best to immediately set its size. 3) The primary panel comes next; all the secondary panels are added to the main panel.

4) **Adding elements to the grid of any panel moves from the top left column to the right column on the first row. After it finishes the first row it moves down a row and repeats the process until all the columns in each row has been completed.**
5) All the secondary panels are added to the primary panel.

   As a rule of thumb, the number of panels that you need to create is equal to the number of rows in the primary panel. It might not always be the case but it's usually good to start this way. Each secondary panel has its own grid layout. Adding elements to the secondary panels is also dependent on the above rule.

6) Adding labels and text fields are different. Labels have a default text that cannot change while text fields have text that can be changed at will. Creating them and adding to them specific panels is quite easy.

7) Creating and adding buttons is also very easy as it follows the same process as adding labels. However, programming their functionality is a bit tedious as it requires the use of an action listener.

8) The menu bar is a vital GUI element. Within the menu bar there are Menus, and within the menu's there are menu items or even more in depth menus with more menu items. Menu items are similar to buttons in which they must also be programmed to do something specific using an action listener exactly like buttons.

9) It is vital that at the end of the program to set the frame to visible= true so that everything will appear.

GUI program format

```
public class classname
{
Declarations….

public classname()
{
makeframe(); // calling a user defined function makeframe
within the constructor.
}
public void makeframe()
{
```

```
GUI code goes here……
}
}
```

There are two ways to create action listeners for buttons and menu items.

The first way is the easiest way and is inserted within the makeframe function.

```
GUIelement.addActionListener(new ActionListener() {

        public void actionPerformed(ActionEvent ae)
        {
           code that allows the button/menu item to function
goes here……
        }
     });
```

Or

We create another function for each button that allows the button/menu item to work properly.

```
public void actionPerformed(ActionEvent ae)     {
      if(ae.getSource().equals(myButton))//this in built
function checks whether the mouse click is coming from a
certain button/menu item. (In this case it is a button called
myButton).
      {
         Do something……
        }
        else
      {
         Do another thing……
        }

   }
```

Create a GUI program that decrements and increments integer numbers with the use of two buttons.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

```
public class Example4
{
   private JFrame myFrame;
   private JLabel myLabel;
   private JButton myButton;
   private JButton mybutton;
   private int count;

   public Example4()
   {
     makeFrame();
   }

   private void makeFrame()
   {
      myFrame = new JFrame("Example 4");
      myFrame.setSize(350,200);

myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
;

      myLabel = new JLabel("Count value: "+count);
      myButton = new JButton("increment");
      mybutton= new JButton("decrement");


      myButton.addActionListener(new ActionListener()
      {
         public void actionPerformed(ActionEvent ae) {
            count++;
            myLabel.setText("Count value: "+ count);
         }
      });
      mybutton.addActionListener(new ActionListener()
      {
         public void actionPerformed(ActionEvent ae) {
            count--;
            myLabel.setText("Count value: "+ count); //you can
put entire variable within the set text in labels or any other
GUI element.
         }
      });
      JPanel panel = new JPanel();
      panel.add(myLabel);
      panel.add(myButton);
      panel.add(mybutton);
      myFrame.getContentPane().add(panel);
```

```
                myFrame.setVisible(true);
            }
        }
```

Question: Create a Java GUI program that displays or hides the first name and last name with the click of two individual buttons.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class button
{
   private JFrame frame;
   private JPanel panel;
   private JPanel panel1;
   private JPanel panel2;
   private JLabel label;
   private JLabel label1;
   private JButton button;
   private JButton button2;
   public button()
   {
      makeframe();
   }

 private void makeframe()
   {
      JFrame frame= new JFrame("Assignment 5");
      frame.setSize(600,200);

      JPanel panel= new JPanel();
      JPanel panel1= new JPanel();
      JPanel panel2= new JPanel();

      panel.setLayout(new GridLayout(2,2));
      panel1.setLayout(new GridLayout(1,3));
      panel2.setLayout(new GridLayout(1,3));


      JLabel label= new JLabel("First Name");
      JLabel label1= new JLabel("Family Name");

      panel1.add(label);
      panel2.add(label1);
```

```
        JTextField text1= new JTextField("Adnan");
        JTextField text2= new JTextField("Ounis");
        panel1.add(text1);
        panel2.add(text2);

        JButton button= new JButton("Hide/Show");
        JButton button2= new JButton("Hide/Show");

       panel1.add(button);
       panel2.add(button2);

       panel.add(panel1);
       panel.add(panel2);


      panel.add(panel2);

      button.addActionListener(new ActionListener() {

           public void actionPerformed(ActionEvent ae)
           {
               if(text1.isVisible())
              text1.setVisible(false);
             else
              text1.setVisible(true);

           }
        });

         button2.addActionListener(new ActionListener() {

           public void actionPerformed(ActionEvent ae)
           {
               if(text2.isVisible())
              text2.setVisible(false);
             else
              text2.setVisible(true);

           }
        });
        frame.getContentPane().add(panel);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

	Question: Create a Full GUI program that shows the first name, last name and country of origin for a specific person. The GUI should also be able to save the data, cancel any new entries, show the inserted entries, set the initial entries and delete any entry.
	import javax.swing.*;
	
	import java.awt.*;
	
	import java.awt.event.*;
	
	import java.awt.Menu;
	public class button
	{
	   private JLabel firstname;
	
	   private JLabel lastname;
	
	   private JLabel country;
	
	   private JButton save;
	
	   private JButton cancel;
	
	   private JButton print;
	
	   private JMenuBar menubar;
	
	   private JMenu data;
	
	   private JMenu about;
	
	   private JMenuItem setinitialdata;
	
	   private JMenuItem deletedata;
	
	   private JMenuItem about1;
	
	   private JFrame lb11;
	
	   private JPanel panel;
	
	   private JPanel panel1;
	
	   private JPanel panel2;
	
	   private JPanel panel3;
	
	   private JPanel panel4;

```
private JPanel panel5;

private String s1,s2,s3;
public button()

{

   makeframe();

}

private void makeframe()

{

   JFrame lb11= new JFrame("lb11");

   lb11.setSize(600,200);

   lb11.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);



   JPanel panel= new JPanel();

   JPanel panel1= new JPanel();

   JPanel panel2= new JPanel();

   JPanel panel3= new JPanel();

   JPanel panel4= new JPanel();

   JPanel panel5= new JPanel();


    panel.setLayout(new GridLayout(5,1));

    panel2.setLayout(new GridLayout(1,2));

    panel3.setLayout(new GridLayout(1,2));

    panel4.setLayout(new GridLayout(1,2));

    panel5.setLayout(new GridLayout(1,3));
```

```
JLabel firstname= new JLabel("First Name:");

JLabel lastname= new JLabel("Last Name:");

JLabel country= new JLabel("Country:");


JTextField text1= new JTextField("Biblo");

JTextField text2= new JTextField("Baggins");

JTextField text3= new JTextField("Shire");


JButton save= new JButton("Save");

JButton cancel= new JButton("Cancel");

JButton print= new JButton("Print");


JMenuBar menubar= new JMenuBar();

JMenu data = new JMenu("Data");

JMenu about= new JMenu("About");

menubar.add(data);

menubar.add(about);

JMenuItem setinitialdata= new JMenuItem("Set Initial Data");

JMenuItem deletedata= new JMenuItem("Delete Data");

JMenuItem about1= new JMenuItem("about");

data.add(setinitialdata);

data.add(deletedata);

about.add(about1);
```

```
 about1.addActionListener(new ActionListener() {

           public void actionPerformed(ActionEvent ae)
           {
              JOptionPane.showMessageDialog(null,"Eggs are not
supposed to be green.");

           }
       });

save.addActionListener(new ActionListener(){
             public void actionPerformed(ActionEvent ae)
             {
             s1=text1.getText();
             s2=text2.getText();
             s3=text2.getText();
             }
             });


             setinitialdata.addActionListener(new ActionListener(){
             public void actionPerformed(ActionEvent ae)
             {
   text1.setText("Bilbo");
             text2.setText("baggins");
             text3.setText("shire");
             }
             });

             deletedata.addActionListener(new ActionListener(){
             public void actionPerformed(ActionEvent ae)
             {
             text1.setText("");
             text2.setText("");
             text3.setText("");
             }
             });

             print.addActionListener(new ActionListener(){
             public void actionPerformed(ActionEvent ae)
             {
             JOptionPane.showMessageDialog(null, "First Name:\t" +
text1.getText() + "\n" + "Last Name: \t" + text2.getText() + "\n" +
"Country: \t" + text3.getText());
             }
             });
```

```
            cancel.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent ae)
            {
    text1.setText("Bilbo");
            text2.setText("baggins");
            text3.setText("shire");
            }
            });

        panel.add(menubar);

        panel2.add(firstname);

        panel3.add(lastname);

        panel4.add(country);




        panel5.add(cancel);

        panel5.add(print);

        panel5.add(save);


        panel2.add(text1);

        panel3.add(text2);

        panel4.add(text3);

        panel.add(panel2);

        panel.add(panel3);

        panel.add(panel4);

        panel.add(panel5);

        lb11.getContentPane().add(panel);

        lb11.setVisible(true);

    }

}
```

Image Processing Questions:

Create an image filter that colors the image.

```
int height = getHeight();
int width = getWidth();
for(int y = 0; y < height; y++)
{
for(int x = 0; x < width; x++)
 {
Color pixel = getPixel(x, y);
//alter the pixel's color value;
setPixel(x, y, pixel);
}
}
```

Create an image filter that lightens the image.

```
private void makeLighter()
{
 if(currentImage != null)
{ currentImage.lighter();
frame.repaint();
showStatus("Applied: lighter");
 } else
 {
showStatus("No image loaded.");
}
 }
```

Create an image filter that segments the image

```
private void threshold()
{
if(currentImage != null)
{
currentImage.threshold();
frame.repaint();
showStatus("Applied: threshold");
}
else
{
showStatus("No image loaded.");
}
}
```

	Exception Handling
Definition	Exceptions are designed to handle errors in your code. So what are errors? 1) Errors are conditions which cause our program to terminate. 2) Sometimes we want to detect and handle the error in the code, without the need to terminate the program. There are two main types of errors that can occur. a) User errors: Such as inputting wrong things or doing illegal actions. b) Programmer errors: Bugs in program.
Advantages	Helps the programmer anticipate any problems that might occur with code and the fix can be implemented directly in the code.
Disadvantages	N.A
Imported Library	Note: All exceptions are of type Exception. Each exception that will be used to handle the error should be imported. For example: If I want to use the InputMismatchException to handle an anticipated error then I would have to include this library: Import java.util.InputMismatchException;
Exception Theory	1) In Java, when an error is detected an exception is thrown. 2) Exceptions can be caught and handled with try-catch code block. 3) It's much cleaner and more effective than looking and checking for the error yourself.
Exception Examples	IndexOutOfBoundsException : an index into an array is out of bounds. •IllegalArgumentException: a method has been called with an invalid argument. •EOFException: an end-of-file mark has been seen. •ArithmeticException : something, such as division by zero, has gone wrong in an arithmetic expression •NumberFormatException : indicates that an illegal number format is being used. •NullPointerException : a class method is being called by an object instance that is currently null.
Common Built-in Functions and their uses	In order to catch and handle these errors within our code we will use a new code block called the try-catch block. The code that you think is susceptible to errors is put within the try-catch block.

	The syntax is as follows: -
	try
	{
	//some code that has the potential to throw an exception.
	}
	catch (Exception e)
	{
	// if an exception is thrown from the above code then we the program immediately comes here rather than terminate. In this section of the block is the code that handles the exception.
	}
	finally
	{
	// code that we put regardless if an exception is thrown or caught. This block always executes.
	}

Method Implementation	1) Write a Java program that allows the user to input an integer number and is then outputted to the screen. (Use exceptions)

```
import java.util.InputMismatchException;
import java.util.Scanner;
public class Exception
{
  int i;
  boolean done= false;

  public Exception()
  {
    number();
  }
  public void number( )
  {
    System.out.println("Please Insert an integer Number.");
    Scanner input= new Scanner(System.in);
    while(!done)
    {
    try
    {
       i= input.nextInt();
       done=true;

    }
    catch(InputMismatchException e)
    {
      System.out.println("Please insert an integer number.");
      input.nextLine();
    }
}
System.out.println("The number is:\t"+i);

    }
}
```

2) Write a Java program that wants the user to insert an integer number between a given specified range. You must create your own exception for the range problem.

In order to create your own exception, you must create a separate class that extends either the Exception or RuntimeException classes.

The big question is when to extend either of the classes when creating your own exception?

The answer all comes down to whether the exception you are creating is an unchecked or checked exception.

Checked exceptions: Checked exceptions are checked at compile-time. It means if a method is throwing a checked exception then it should handle the exception using try-catch block or it should declare the exception using throws keyword, otherwise the program will give a compilation error. It is named as ***checked exception*** because these exceptions are ***checked*** at Compile time. Checked exceptions extend the Exceptions class.

Unchecked exceptions: Unchecked exceptions are not checked at compile time. It means if your program is throwing an unchecked exception and even if you didn't handle/declare that exception, the program won't give a compilation error. Most of the times these exception occurs due to the bad data provided by user during the user-program interaction. It is up to the programmer to judge the conditions in advance, that can cause such exceptions and handle them appropriately. All Unchecked exceptions are direct sub classes of **RuntimeException** class.

```
//Custom exception class
public class rangeexception extends RuntimeException
{
   public rangeexception(String r)
   {
      super(r);
   }
}
```

```
//Java program that requires my exception.
import java.util.InputMismatchException;
import java.util.Scanner;
public class Exception
{
   int i;
   boolean done= false;

   public Exception()
   {
      number();
   }
   public void number( )
{
      System.out.println("Please Insert an integer Number
between 1 and 10.");
      Scanner input= new Scanner(System.in);
      while(!done)
      {
         try
         {
          i= input.nextInt();
          done=true;

          if(i<1 || i>10)
          throw new rangeexception("invalid");
//you must put a parameter as the rangeexception class
takes a parameter of type string.
         }
         catch(InputMismatchException e)
         {
         System.out.println("Please insert an integer
number.");
         input.nextLine();
         }
         catch(rangeexception e)
         {
         System.out.println("Please insert an integer number
between the specified range.");
         done=false;
         input.nextLine();
         }

      }
 System.out.println("The number is:\t"+i);
}

}
```

Question 3)

Implement a class SimpleAccount with the following characteristics:
1.fields: number of type String and balance of type float

2.a constructor taking the account number and setting the balance to 0

3.getter methods for both fields

4.a setter method for balance

5.a method print() that prints the values of both fields

6.a method interest(int percentage) that adds percentage * balance to the account balance and returns the new balance value

if the percentage is < 0 throw a checked exception with the text (invalid interest rate)

if the percentage is > 10 throw an unchecked exception with the text (information, very high interest rate)

if the percentage is > 100 throw a checked exception with the text (interest rates of more than 100 are not allowed)

Answer:

```
//exception class 1
public class percentageexception extends Exception
{
   public percentageexception()
   {

   }
}
//exception class 2
public class percentageexception1 extends Exception
{
   public percentageexception1()
   {

   }
}
```

```
//exception class 3
public class percentageexception2 extends RuntimeException
{
   public percentageexception2()
   {

   }
}
```
Main java program:
```
public class SimpleAccount
{
   private String number;
   private float balance;

   public SimpleAccount(String n)
   {
      number=n;
      balance=0;
   }
   public String getNumber()
   {
      return number;
   }
   public float getBalance()
   {
      return balance;
   }
   public void setBalance(float b)
   {
      balance=b;
   }
   public void print()
   {
      System.out.println("Your account number is:\t" +number);
      System.out.println("Your current balance is: \t" +balance);
   }
public float interest(float percentage)
   {
      try
      {
      if(percentage>0 && percentage<10)
      {
       balance= balance+(balance*(percentage/100));
      }
      if(percentage<0)
      {
        throw new percentageexception();
      }
      if (percentage>10)
      {
        throw new percentageexception1();
      }
      if (percentage>100)
      {
        throw new percentageexception2();
      }
   }
   catch(percentageexception e)
   {
      System.out.println("Invalid interest rate");
   }
   catch(percentageexception1 e)
   {
      System.out.println("too high interest rate");

   }
   catch(percentageexception2 e)
   {
      System.out.println("Interest rate must be between 0 and 10");
   }

      return balance;
   }
}
```