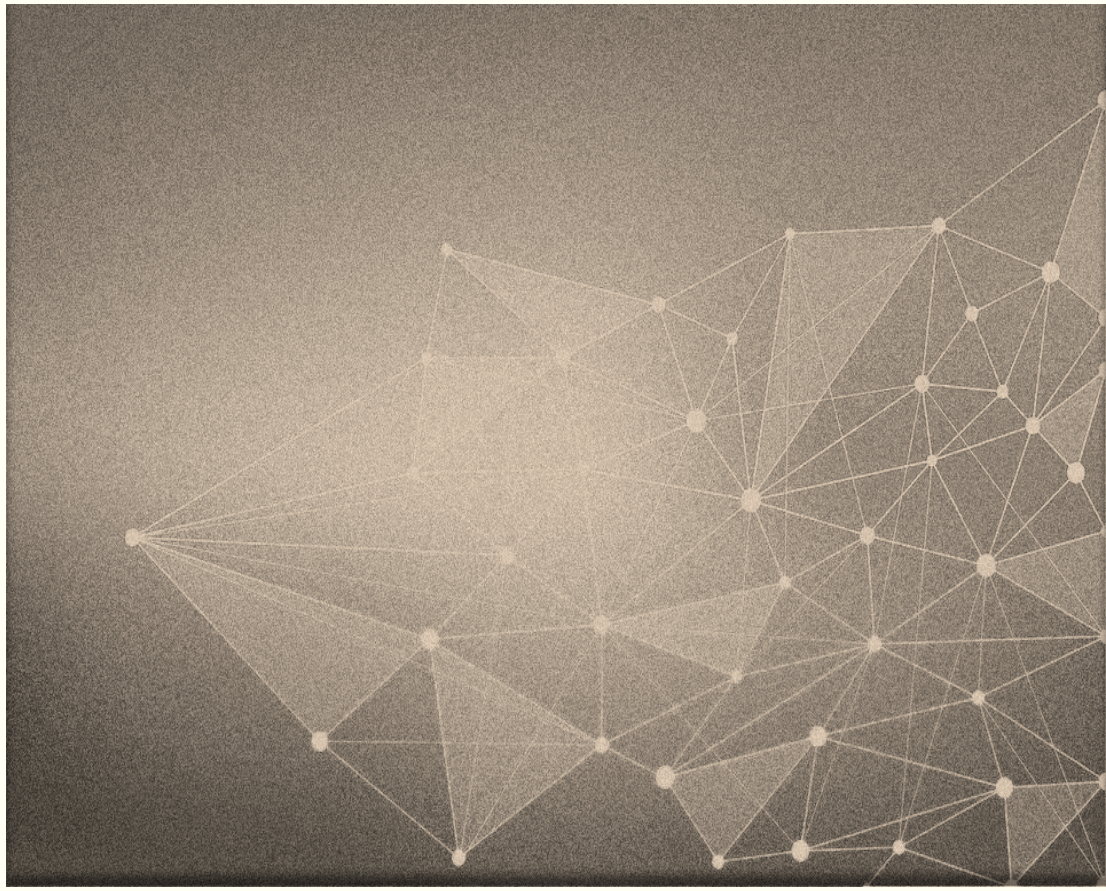
EECS2030:
ADVANCED OBJECT-ORIENTED PROGRAMMING

By: Dr. Marzieh Ahmadzadeh



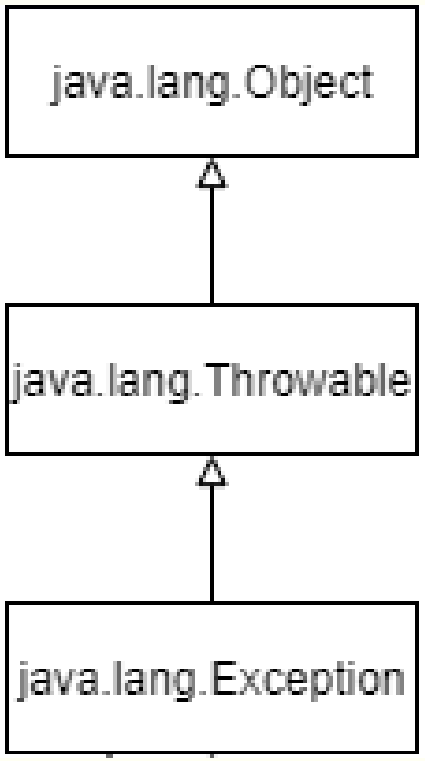
Outline

- Last Lecture:
  - Polymorphism
  - Binding
  - DBC

- This week:
  - Exception handling
  - Object class
    - toString() revisited.
    - equals()
    - hashCode()
  - Comparable and comparTo()

Exceptions

- Recall: Two types of exceptions:
  - Unchecked
  - Checked
- How to handle unchecked exceptions?



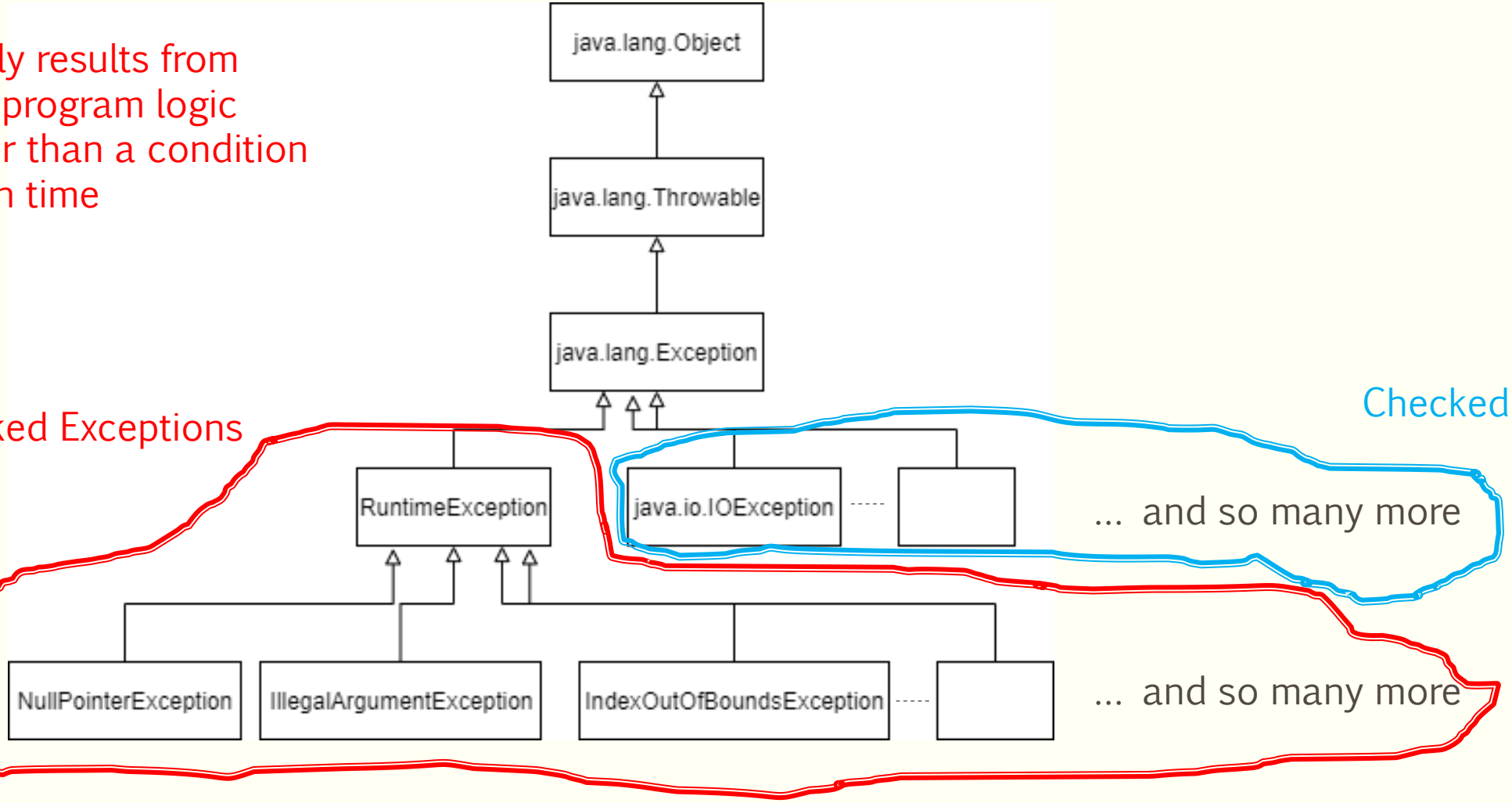
- In java, Exception is a class that inherits from Throwable.
- Only an instance of a Throwable (or its subclasses) can be thrown by the JVM.
- Only an instance of a Throwable(or its subclass) can be the argument in a catch clause.

Exception Hierarchy

Mostly results from your program logic rather than a condition at run time

Unchecked Exceptions

Checked Exception



What if an exception happens?

Unchecked

- If you handled it:
  - A customized version of the exception will be shown.
- If you don't not handle it
  - Java will take care of the error

```
public void wrongMethod1 () {
    ArrayList<Integer> arrayObj = new ArrayList<Integer>();
    System.out.println(arrayObj.get(0));
}
```

Checked

- You have already handled it as Java enforced you to do it.
  - A customized version of the exception is shown.

How to handle unchecked exceptions?

- Get the method to throw an exception:

```
public void wrongMethod2() throws IndexOutOfBoundsException{
    ArrayList<Integer> arrayObj = new ArrayList<Integer>();
    System.out.println(arrayObj.get(0));
}
```

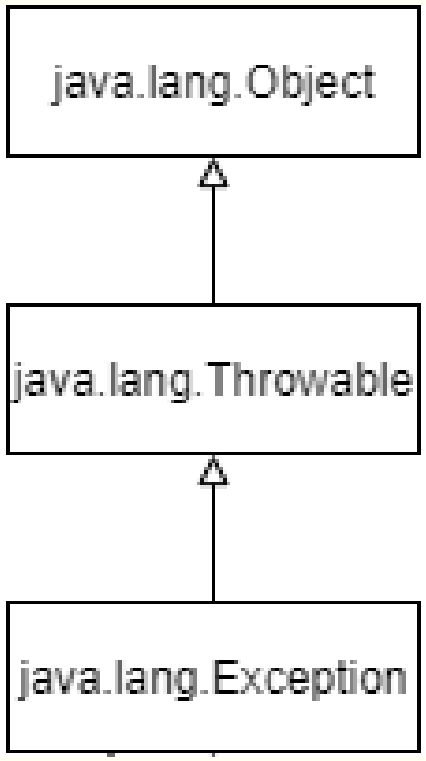
- Did we do anything useful here?

How to make our own exception?

- We want to be more specific about the errors in order to debug it faster.

- Create your own Exception class.
  - Set the IS-A relationship between your exception class and one of the java exception classes.
  - Override some of the constructors.

- Methods from Object, Throwable and Exception classes are inherited to your exception class.



Customized Exception

```java
class NegativeNumberException extends Exception{
    public NegativeNumberException (){
        super();
    }
    public NegativeNumberException(String message){
        super(message);
    }
}
```

```java
class NumberZeroException extends Exception{
    public NumberZeroException (){
        super();
    }
    public NumberZeroException(String message){
        super(message);
    }

}
```

Throw & Catch a user-defined exception

- You can throw as many exceptions as you require.

```
public void printMonth(int month){
    try {
        if (month < 0 ) throw new NegativeNumberException("a negative number is not accepted as month!");
        if (month == 0) throw new NumberZeroException("Zero is not accepted as a month!");
        // insert the code here
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```

Polymorphism

Inherited from Throwable

Throwing without catching it.

- You can throw as many exception as you require.
- In method signature add: `throws` *`name_of_the_exception`*

```
public String getMonth(int month) throws NegativeNumberException, NumberZeroException{
    if (month < 0 ) throw new NegativeNumberException("a negative number is not accepted as month!");
    if (month == 0) throw new NumberZeroException("Zero is not accepted as a month!");
    // insert the code here
    return "";
}
```

- What is the difference between these two methods (throwing with/without catching in the same method)?

User-defined vs Java Exceptions

- Exception handling is different in user-defined exception from java exceptions.

```java
public void wrongMethod2() throws IndexOutOfBoundsException{
    ArrayList<Integer> arrayObj = new ArrayList<Integer>();
    System.out.println(arrayObj.get(0));
}
```

Java unchecked Exception

```java
public String getMonth(int month) throws NegativeNumberException, NumberZeroException{
    if (month < 0 ) throw new NegativeNumberException("a negative number is not accepted as month!");
    if (month == 0) throw new NumberZeroException("Zero is not accepted as a month!");
    // insert the code here
    return "";
}
```

User-defined Exception

Note: for java checked exception, you must use try-catch structure

Exception and Inheritance

- If an overridden method throws an exception, the super method also must throw the same or higher level of exception (Polymorphism).
- Remember that a subclass should be able to do whatever the superclass can do even more.

```
class A {
    public void method1 () throws NumberZeroException{}
    public void method2 () throws NumberZeroException{}
    public void method3 () {}
    public void method4 () throws Exception{}
    public void method5 () throws NumberZeroException{}
}

class B extends A{
    public void method1 () throws NumberZeroException {}
    public void method2 (){}
    public void method3 () throws NumberZeroException{}
    public void method4 () throws NumberZeroException{}
    public void method5 () throws Exception{}
}
```

A few tips

- A method that throws an exception in the method signature, can throw any subclass of the exception inside the methods.
  - Polymorphism

- A method can throw several exceptions
  - As seen in the previous examples

- A method can catches more than one exceptions.

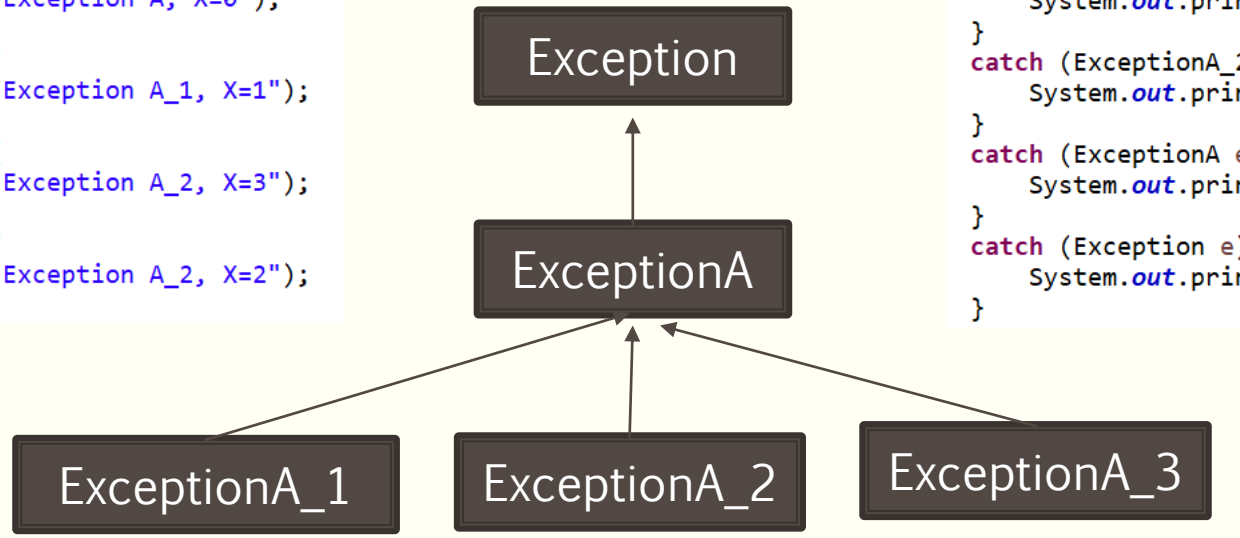
```
        // insert the code here
    } catch (NegativeNumberException e) {
        System.out.println(e.getMessage());
    }
    catch (NumberZeroException e) {
        System.out.println(e.getMessage());
    }
```

Order of Catch Statement

- Compare the following two codes:

```java
try {
    if (x == 3) throw new ExceptionA_3();
    if (x == 2) throw new ExceptionA_2();
    if (x == 1) throw new ExceptionA_1();
    if (x == 0) throw new ExceptionA();
    if (x < 0 ) throw new Exception();
}
catch (Exception e) {
    System.out.println("Exception, x < 0");
}
catch (ExceptionA e) {
    System.out.println("Exception A, X=0");
}
catch (ExceptionA_1 e) {
    System.out.println("Exception A_1, X=1");
}
catch (ExceptionA_3 e) {
    System.out.println("Exception A_2, X=3");
}
catch (ExceptionA_2 e) {
    System.out.println("Exception A_2, X=2");
}
```

```java
try {
    if (x == 3) throw new ExceptionA_3();
    if (x == 2) throw new ExceptionA_2();
    if (x == 1) throw new ExceptionA_1();
    if (x == 0) throw new ExceptionA();
    if (x < 0 ) throw new Exception();
}
catch (ExceptionA_1 e) {
    System.out.println("Exception A_1, X=1");
}
catch (ExceptionA_3 e) {
    System.out.println("Exception A_2, X=3");
}
catch (ExceptionA_2 e) {
    System.out.println("Exception A_2, X=2");
}
catch (ExceptionA e) {
    System.out.println("Exception A, X=0");
}
catch (Exception e) {
    System.out.println("Exception, x < 0");
}
```



Activity

- Q1

- Exception handling does not necessarily result in program termination.

```java
import java.util.Scanner;
import java.util.InputMismatchException;

public class Q1 {
    public static void main(String [] args) {

        int num = getNumber();
        System.out.println("The entered number is: " + num);
    }
    public static int getNumber() {
        Scanner sc = new Scanner(System.in);
        int input = 0;
        boolean finished = false;
        while (!finished) {
            try {
                System.out.print("Enter a whole number:");
                input = sc.nextInt();
                finished = true;
            }
            catch (InputMismatchException e) {
                sc.nextLine();
                System.out.println("Wrong Input, try again...");

            }
        }

        return input;
    }
}
```

```java
import java.util.InputMismatchException;
import java.util.Scanner;

public class Q3 {
    public static void main(String [] args) {

        int num = getNumber();
        System.out.println("The entered number is: " + num);
    }
    public static int getNumber() {
        Scanner sc = new Scanner(System.in);
        int input = 0;
        boolean finished = false;
        while (!finished) {
            try {
                System.out.print("Enter a whole number:");
                input = sc.nextInt();
                finished = true;
            }
            catch (InputMismatchException e) {
                sc.nextLine();
                System.out.println("Wrong Input, try again...");
                System.exit(0);
            }
        }

        return input;
    }
}
```

OBJECT CLASS

Object Class

- Recall:
  - Object is the ancestor of all the classes in java.
  - It is in java.lang, so there is no need to import it
  - Its methods are inherited to all java classes.

- We only study some of them.
  - equlas(), hashCode(), toString()

- We call them obligatory methods, as you need to override these methods to fit your purpose.

Constructor	Description
`Object()`	Constructs a new object.

Method Summary

All Methods | Instance Methods | Concrete Methods | Deprecated Methods

Modifier and Type	Method	Description
`protected Object`	`clone()`	Creates and returns a copy of this object.
`boolean`	`equals(Object obj)`	Indicates whether some other object is "equal to" this one.
`protected void`	`finalize()`	**Deprecated.** The finalization mechanism is inherently problematic.
`Class<?>`	`getClass()`	Returns the runtime class of this `Object`.
`int`	`hashCode()`	Returns a hash code value for the object.
`void`	`notify()`	Wakes up a single thread that is waiting on this object's monitor.
`void`	`notifyAll()`	Wakes up all threads that are waiting on this object's monitor.
`String`	`toString()`	Returns a string representation of the object.
`void`	`wait()`	Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*.
`void`	`wait (long timeoutMillis)`	Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*, or until a certain amount of real time has elapsed.
`void`	`wait (long timeoutMillis, int nanos)`	Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*, or until a certain amount of real time has elapsed.

Recall: toString()

- The original implementation returns the following, which basically is the address of the object in hash table.
  - Stay tuned to see what is a hash table

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

```
System.out.println(myDog);
System.out.println(myDog.toString());
```

```
Week8.Dog@5ca881b5
Week8.Dog@5ca881b5
```

Object.toString(), Not a useful information

- It is suggested that you override the method.

```
@Override
public String toString() {
    return "This is a dog named " + name + ", whose picture can be found in " + picturePath;
}
```

equals()

- The Object's equals() checks for the equality of `object references`.

```
Dog myDog = new Dog("Rosie", "C:/pictures/rosie.jpg");
Dog herDog = new Dog("Rosie", "C:/pictures/rosie.jpg");
Dog yourDog = myDog;

System.out.println (yourDog.equals(myDog));
System.out.println (herDog.equals(myDog));
```

- You should override the method if you expect a different functionality.

`boolean`	`equals(Object obj)`	Indicates whether some other object is "equal to" this one.

Example 1:

- This example show you how equals() work, when the class has only primitive attributes

```
class OnlyPrimitives{
    int firstAttribute;
    char secondAttribute;
    public OnlyPrimitives() {
        firstAttribute = 0;
        secondAttribute = ' ';
    }
    public OnlyPrimitives(int first, char second) {
        firstAttribute = first;
        secondAttribute = second;
    }
}
```

```
OnlyPrimitives obj1 = new OnlyPrimitives(10, 'A');
OnlyPrimitives obj2 = new OnlyPrimitives(10, 'A');
OnlyPrimitives obj3 = obj1;
System.out.println (obj1.equals(obj2));
System.out.println (obj1.equals(obj3));
```

Example 2:

- This example show you how equals() work, when the class has simple non-primitive attributes.

```
class NonPrimitives{
    int firstAttribute;
    String secondAttribute;
    public NonPrimitives() {
        firstAttribute = 0;
        secondAttribute = "";
    }
    public NonPrimitives(int first, String second) {
        firstAttribute = first;
        secondAttribute = new String(second);
    }
}
```

```
NonPrimitives obj4 = new NonPrimitives(10, "A");
NonPrimitives obj5 = new NonPrimitives(10, "A");
NonPrimitives obj6 = obj4;
System.out.println (obj4.equals(obj5));
System.out.println (obj4.equals(obj6));
```

Example 3:

- This example show you how equals() work, when the class has complex non-primitive attributes.

```
class ComplexNonPrimitives{
    int firstAttribute;
    ArrayList<String> secondAttribute;
    public ComplexNonPrimitives() {
        firstAttribute = 0;
        secondAttribute = new ArrayList<String>();
    }
    public ComplexNonPrimitives(int first, ArrayList<String> second) {
        firstAttribute = first;
        secondAttribute = new ArrayList<String>();
        for (String obj:second) {
            secondAttribute.add(new String(obj));
        }
    }
}
```

```
ArrayList<String> arr = new ArrayList<String>();
arr.add("A");
arr.add("B");
ComplexNonPrimitives obj7 = new ComplexNonPrimitives(10, arr);
ComplexNonPrimitives obj8 = new ComplexNonPrimitives(10, arr);
ComplexNonPrimitives obj9 = obj7;
System.out.println (obj7.equals(obj8));
System.out.println (obj7.equals(obj9));
```

Implementation of equals()

- Object's equals() check if the two objects references point to the same object.
- What if you want to see if the attributes of an object have similar values?
  - Override equals()
  - The definition of equality depends on your application
- You can override equalTo() in anyways providing that the following requirements are met:
  - For any non null reference x, y and z,
    - Reflexive property: x.equals(x) -> true
    - Symmetric property: x.equals(y) -> true iff y.equals(x) -> true
    - Transitive property: if x.equals(y) -> true & y.equals(z) -> true ==> x.equals(z) -> true
    - Consistency: multiple invocation of x.equal(y) return the same thing if x and y do not change.
    - x.equals(null) -> false

Example 1:

- This example shows you how equals() is implemented, when the class has only primitive attributes

```
public OnlyPrimitives(int first, char second) {
    firstAttribute = first;
    secondAttribute = second;
}
```

```
public boolean equals (Object object) {
    OnlyPrimitives obj = (OnlyPrimitives) object;
    if (firstAttribute == obj.firstAttribute && secondAttribute == obj.secondAttribute)
        return true;
    else return false;
}
```

```
OnlyPrimitives obj1 = new OnlyPrimitives(10, 'A');
OnlyPrimitives obj2 = new OnlyPrimitives(10, 'A');
OnlyPrimitives obj3 = obj1;
System.out.println (obj1.equals(obj2));
System.out.println (obj1.equals(obj3));
```

Q: what if object is null?

Check this implementation against the requirements discussed in slide 26.

Example 2:

- This example shows you how equals() is implemented, when the class has primitive and simple non-primitive attributes

```
public NonPrimitives(int first, String second) {
    firstAttribute = first;
    secondAttribute = new String(second);
}
```

```
public boolean equals (Object object) {
    NonPrimitives obj = (NonPrimitives) object;
    boolean equal = false;
    if (obj != null)
        if (firstAttribute == obj.firstAttribute && secondAttribute.compareTo(obj.secondAttribute) == 0)
            equal = true;
    return equal;
}
```

```
NonPrimitives obj4 = new NonPrimitives(10, "A");
NonPrimitives obj5 = new NonPrimitives(10, "A");
NonPrimitives obj6 = obj4;
System.out.println (obj4.equals(obj5));
System.out.println (obj4.equals(obj6));
```

Check this implementation against the requirements discussed in slide 26.

Example 3:

- This example shows you how equals() is implemented, when the class has complex non-primitive attributes

```java
public ComplexNonPrimitives(int first, ArrayList<String> second) {
    firstAttribute = first;
    secondAttribute = new ArrayList<String>();
    for (String obj:second) {
        secondAttribute.add(new String(obj));
    }
}
```

```java
public boolean equals (Object object) {
    ComplexNonPrimitives obj = (ComplexNonPrimitives) object;
    boolean equal = (obj != null && this.secondAttribute.size() == obj.secondAttribute.size() &&
                     this.firstAttribute == obj.firstAttribute);
    if (equal)
        for (int i = 0; i < obj.secondAttribute.size(); i++)
            if (secondAttribute.get(i).compareTo(obj.secondAttribute.get(i)) != 0) {
                equal = false;
                break;
            }
    return equal;
}
```

```java
ArrayList<String> arr = new ArrayList<String>();
arr.add("A");
arr.add("B");
ComplexNonPrimitives obj7 = new ComplexNonPrimitives(10, arr);
ComplexNonPrimitives obj8 = new ComplexNonPrimitives(10, arr);
ComplexNonPrimitives obj9 = obj7;
System.out.println (obj7.equals(obj8));
System.out.println (obj7.equals(obj9));
```

Check this implementation against the requirements discussed in slide 26.

Missing code

- Something is missing in all the previous codes. Do you know what is it?
- How do you check if the type of two objects is the same?

```
this.getClass() != obj.getClass()
```

Application of equals

- **`equals()`** is a very useful method for searching techniques.
- Example: An ArrayList of all enrolled students in EECS2030 is given:
  - Find if "John" has enrolled in this class.
  - Find all the students, who have enrolled late and were not able to submit the early assignments.
  - Update the grade of the student, whose name is "Jane"
- These are an example of linear searching.
- How much money should I spend, if I am obliged to put 1 dollar in a piggy bank for every use of equals()?
  - Best case?
  - Worst case?
  - Average case?

Hash Tables

- Is a data structure that lets you search faster (i.e. less usage of equals())
- A hash table consists of
  - A bucket array of size N
  - hash function h
- A hash function h maps the input data to integers in a fixed interval [0, N - 1]
  - e.g. $h(k) = k \bmod N$
  - h(k) is called the `hash code` / `hash value` of k
  - k is stored at index h(k)
- Example: h = k mod 5,
  - Insert 6, 7, 21, 9, 10, 6
  - Is 9 exists in the array?
  - Remove 9
- How much money have you saved?

Object's HashCode() function

- hashCode returns the memory address in which the object is stored.

```
class OnlyPrimitives{
    int firstAttribute;
    char secondAttribute;
```

```
OnlyPrimitives obj1 = new OnlyPrimitives(10, 'A');
OnlyPrimitives obj2 = new OnlyPrimitives(10, 'A');
OnlyPrimitives obj3 = obj1;
```

```
System.out.println(obj1.hashCode() + "\t" + obj2.hashCode() + "\t" + obj3.hashCode());
System.out.println(obj1);
```

The same memory address in decimal

Different from the other two, although the objects are the same.

The same memory address but in hexadecimal

int	hashCode()	Returns a hash code value for the object.

Overridden hashCode()

- A hash code must at least satisfies the following contracts:
  - Frequently calling obj.hashCode() `at one execution` should return the same address.
  - obj1.equals(obj2) -> true ➔ obj1.hashCode() == obj2.hashCode()
- It is possible that obj1.equals(obj2) = false, but still obj1.hashCode() == obj1.hashCode()
    - This is not ideal, it is called collision. You'll learn more in Data Structures course
- How to implement the hashCode()?
  - Not an easy job. You want to minimize the collision while satisfying the above contracts.
  - You'll learn about it in detail in Data Structures course
  - But in the meantime
    - use Objects.hash() method

`static int`	`hash(Object... values)`	Generates a hash code for a sequence of input values.
`static int`	`hashCode(Object o)`	Returns the hash code of a non-`null` argument and 0 for a `null` argument.

Overridden hashCode()

```
class OnlyPrimitives{
    int firstAttribute;
    char secondAttribute;
```

```
@Override
public int hashCode() {
    return Objects.hash(firstAttribute, secondAttribute);
}
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    OnlyPrimitives other = (OnlyPrimitives) obj;
    return firstAttribute == other.firstAttribute && secondAttribute == other.secondAttribute;
}
```

```
OnlyPrimitives obj1 = new OnlyPrimitives(10, 'A');
OnlyPrimitives obj2 = new OnlyPrimitives(10, 'A');
OnlyPrimitives obj3 = obj1;
```

- What will be outputted with this new implementation?

```
System.out.println(obj1.hashCode() + "\t" + obj2.hashCode() + "\t" + obj3.hashCode());
System.out.println(obj1);
```

If you're lazy to write these methods, let eclipse generates it for you.
From source menu-> generate hashCode() and equals()

Activity

- Q2

COMPARABLE & COMPARETO()

A step back

- Primitives can be compared by < , > and ==
- For the objects
  - You need to define what the ordering means
  - Implement a method that compares the two objects.

```
public NonPrimitives(int first, String second) {
    firstAttribute = first;
    secondAttribute = new String(second);
}
```

```
public boolean equals (Object object) {
    NonPrimitives obj = (NonPrimitives) object;
    boolean equal = false;
    if (obj != null)
        if (firstAttribute == obj.firstAttribute && secondAttribute.compareTo(obj.secondAttribute) == 0)
            equal = true;
    return equal;
}
```

compareTo()

- As a convention, the comparison method should be called compareTo()

- `CompareTo()` is an abstract method that is defined in `Comparable` Interface.
  - Keep these in mind. It will be explained in next lectures.

- If an order can be defined on objects of X, then X should be Comparable.
- For a class to be comparable, it should implement Comparable Interface and override the only methods that it has.

```
class x implements Comparable{

          public int compareTo(Object obj) {    }

}
```

compareTo()

- If obj1 and obj2 are of type X and comparable then
  - obj1.compareTo(obj2) returns a negative integer, if obj1 < obj2
  - obj1.compareTo(obj2) returns zero if obj1 == obj2
  - obj1.compareTo(obj2) returns a positive integer, if obj1 > obj2

- It throws two exceptions:

Throws:

`NullPointerException` - if the specified object is null

`ClassCastException` - if the specified object's type prevents it from being compared to this object.

Example

- Suppose that a point is said to be less than another, if it is closer to the origin.

```java
class Point implements Comparable <Object>{
    int x;
    int y;

    public Point() {
        this.x = 0;
        this.y = 0;
    }

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public int compareTo(Object p) {
        Point point = (Point) p;
        double firstDist = Math.sqrt(this.x * this.x + this.y + this.y);
        double secondDist = Math.sqrt(point.x * point.x + point.y * point.y);
        return (int) (firstDist - secondDist);
    }
}
```

```java
Point p1 = new Point(2,2);
Point p2 = new Point(3, 4);
Point p3 = new Point(3,4);
System.out.println(p1.compareTo(p2));
System.out.println(p2.compareTo(p1));
System.out.println(p3.compareTo(p2));
```

Comparable Contracts

- If `obj1.compareTo(obj2) < 0` then `obj2.compareTo(obj1) > 0`
- If `obj1.compareTo(obj2) > 0` then `obj2.compareTo(obj1) < 0`
- If `obj1.compareTo(obj2) == 0` then `obj2.compareTo(obj1) == 0`
- If `obj1.compareTo(obj2) < 0 && obj2.compareTo(obj3) < 0` then `obj1.compareTo(obj3) < 0` [Transitivity property]
- If `obj1.compareTo(obj2) > 0 && obj2.compareTo(obj3) > 0` then `obj1.compareTo(obj3) > 0` [Transitivity property]
- If `obj1.compareTo(obj2) == 0 && obj2.compareTo(obj3) == 0` then `obj1.compareTo(obj3) == 0` [Transitivity property]

compareTo() and equals()

- If `obj1.compareTo(obj2) == 0` then `obj1.equals(obj2)` may / may not be true.

- Can you think of a situation where these two methods are not consistent?

- If `obj1.compareTo(obj2) == 0 & obj1.equals(obj2)== true,` then it is said that compareTo() and equals() are consistent.

Expectation and Reading

- Expectation:
  - You should be able to implement your own exception.
  - You should be able to define the difference between checked an uncheck exception.
  - You should be able to throw and catch both the checked and unchecked exceptions.
  - You should fully understand the effect of class Object being the superclass of all objects.
  - You should be able to explain what toString(), hashCode() and equals() defined in class Object, do.
  - You should be able to override toString(), hashCode() and equals() to serve your purpose.
- Reading: While not necessary but you may find the following interesting to read:
  - Exceptions: https://www.tutorialspoint.com/java/java_exceptions.htm
  - Comparable: https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/Comparable.html
  - Object Class: https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/Object.html
- One Minute Paper