Lecture 5 - (OOP II)

Polymorphism

Every instance of a subclass is also an instance of its superclass, but not vice versa

Polymorphism: An object of a subclass can be used wherever its superclass object is used

```
public class Demo {
  public static void main(String [] args) {
    m(new Point(1,2));
  }
  public static void m(Object x) {
    System.out.println(x);
  }
}
```

We can achieve polymorphism in Java using the following ways:

1. Method Overriding

```
class Language {
  public void displayInfo() {
    System.out.println("Common English Language");
  }
}

class Java extends Language {
  @Override
  public void displayInfo() {
    System.out.println("Java Programming Language");
  }
}
```

2. Method Overloading

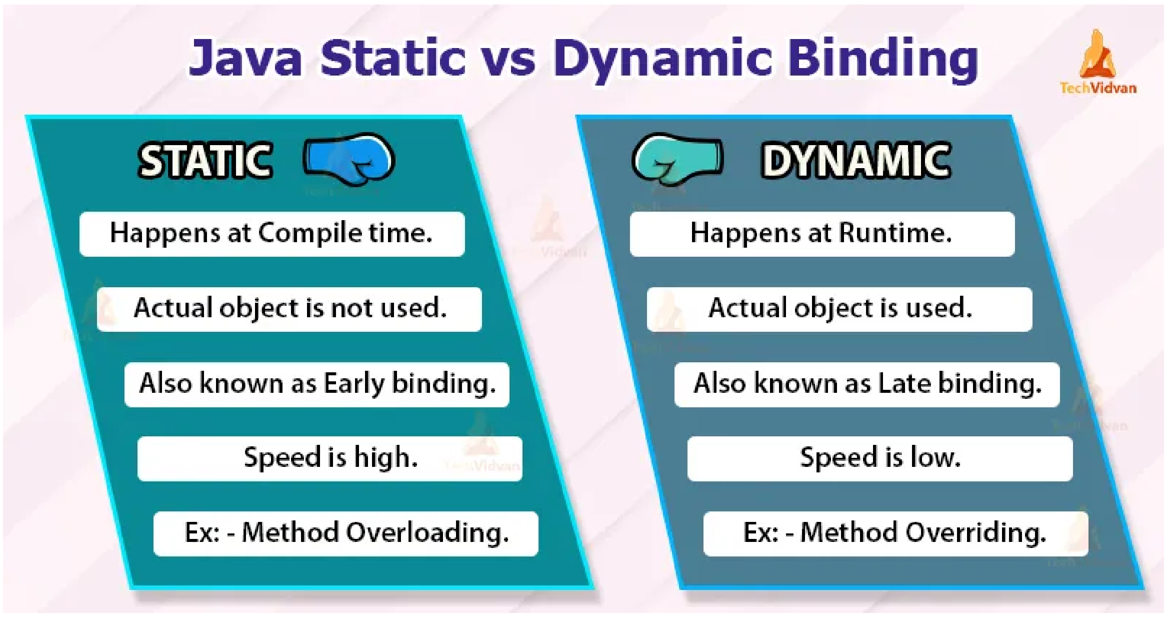
```
void func() { ... }
void func(int a) { ... }
float func(double a) { ... }
float func(int a, float b) { ... }
```

3. Operator Overloading

```
1. When + is used with numbers (integers and floating-point numbers), it performs mathematical addition. For example,

int a = 5;
int b = 6;

// + with numbers
int sum = a + b;  // Output = 11

2. When we use the + operator with strings, it will perform string concatenation (join two strings). For example,

String first = "Java ";
String second = "Programming";

// + with strings
name = first + second;  // Output = Java Programming

Here, we can see that the + operator is overloaded in Java to perform two operations: addition and concatenation.
```

Dynamic Binding

- A method can be implemented in several classes along the inheritance chain



Java Static vs Dynamic Binding

STATIC	DYNAMIC
Happens at Compile time.	Happens at Runtime.
Actual object is not used.	Actual object is used.
Also known as Early binding.	Also known as Late binding.
Speed is high.	Speed is low.
Ex: - Method Overloading.	Ex: - Method Overriding.

TechVidvan

- The JVM dynamically binds the implementation of the method at
  runtime, decided by the actual type of the variable
    - E.g. Object x = new Point(1,2); //declared type: Object, actual type:Point
    - System.out.println(x); //which toString is invoked?
- Dynamic binding works as follows:
    - Suppose an object x is an instance of classes C1, C2, . . . , Cn-1, and Cn, where C1 is a subclass of C2, C2 is a subclass of C3, . . . , and Cn-1 is a subclass of Cn,
    - If x invokes a method p, the JVM searches for the implementation of the method p in C1, C2, . . . , Cn-1, and Cn, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked

  ```
  class Person
  {
    public void speak()
    {
      System.out.println("Person speaks");
    }
  }
  class Teacher extends Person
  {
    @Override
    public void speak()
    {
      System.out.println("Teacher speaks");
    }
  }
  public class DynamicBinding
  {
    public static void main( String args[])
    {
      // Reference and objects are of Person type.
      Person obj2 = new Person();
          obj2.speak();  //Person speaks
      // Reference is of Person type and object is Teacher type
      Person obj = new Teacher();
          obj.speak();   //Teacher speaks
    }
  }
  ```

  From the above code, we got different output because:

    - We have **not declared the methods as static** in the code.
    - During compilation, the **compiler has no idea of which method to call**. This happens because the compiler doesn’t go according to the type of the object but it checks only according to the reference variable. Therefore the binding gets delayed to runtime, so the respective version of the speak() method will be called on the basis of the type of the object
- **Static Binding or Early Binding** in Java refers to a process where the compiler determines the type of object and resolves the method during the compile-time. Generally, the compiler binds the overloaded methods using static binding.

  There is a fact that the binding of static, private, and final methods are always done during compile-time using static-binding.

  ```
  package com.techvidvan.binding;
  class Person
  {
    public void speak()
    {
      System.out.println("Person speaks");
    }
  }
  ```

```
class Teacher extends Person
{
  //error
  // You can’t override a non-static method with a static one, or vice versa.
  public static void speak()
  {
    System.out.println("Teacher speaks");
  }
}
public class StaticBinding
{
  public static void main( String args[ ])
  {
    // Reference is of Person type and object is Teacher type
    Person obj = new Teacher();
        obj.speak();  //Person speaks
    // Reference and object both are of Person type.
    Person obj2 = new Person();
        obj2.speak(); //Person speaks
  }
}
```

From the above code, we got the same output from the parent class. This happened because:

- The **reference for the parent class and the child class is the same(Person).** That is, a single object refers to both of them.
- **Since the method is static,** the **compiler is aware that this method can not be overridden in the child class and it knows which method to call**. Therefore there is no ambiguity and the output is the same for both cases.

Encapsulation

Encapsulation **refers to the bundling of fields and methods inside a single class.**

Encapsulation is one of the key features of object-oriented programming.

- The details of implementation are encapsulated and hidden from the user
- Modules communicate only through their APIs and are oblivious to each other's inner workings
    - Eg using System.out.println without knowing how it is implemented
- Advantages
    - Decoupling the modules that comprise a system allows them to be developed, tested optimized, used, understood and modified in isolation
        - It helps to decouple components of a system. **For example**, we can encapsulate code into multiple bundles.

          These decoupled components (bundle) can be developed, tested, and debugged independently and concurrently. And, any changes in a particular component do not have any effect on other components.
    - **Information hiding** increase software reuse because modules that aren't tightly couple often prove useful in other contexts
    - It helps to **control the values** of our data fields.
    - encapsulation helps us to **keep related fields and methods together**, which makes our code cleaner and easy to read.
    - The **getter and setter** methods provide **read-only** or **write-only** access to our class fields.
    - We can also achieve data hiding using encapsulation. In the above example, if we change the length and breadth variable into private, then the access to these fields is restricted. And, they are kept hidden from outer classes. This is called **data hiding**.

    - It **prevents outer classes from accessing and changing fields and methods of a class**. This also helps to achieve **data hiding**.

- The access control mechanism in Java facilitates encapsulation
- There are four possible access levels for members, listed in order of increasing accessibility:

1. **private**—The member is accessible only from the top-level class where it is declared
2. **package-private**—The member is accessible from any class in the package where it is declared (default access)
3. **protected**—The member is accessible from subclasses of the class where it is declared and from any class in the package where it is declared
4. **public**—The member is accessible from anywhere
   • Rule of thumb: make each member as inaccessible as possible

```
class Area {

  // fields to calculate area
  int length;
  int breadth;

  // constructor to initialize values
  Area(int length, int breadth) {
    this.length = length;
    this.breadth = breadth;
  }

  // method to calculate area
  public void getArea() {
    int area = length * breadth;
    System.out.println("Area: " + area);
  }
}

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

    // create object of Area
    // pass value of length and breadth
    Area rectangle = new Area(5, 6);
    rectangle.getArea();    // Area: 30 printed
  }
}
```

In the above example, we have created a class named Area. The main purpose of this class is to calculate the area.

To calculate an area, we need two variables: length and breadth and a method: `getArea()`. Hence, we bundled these fields and methods inside a single class.

Here, the fields and methods can be accessed from other classes as well. Hence, this is not **data hiding**.

This is only **encapsulation**. We are just keeping similar codes together.

Data Hiding

Data hiding is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding.

We can use access modifiers to achieve data hiding.

```
class Person {

  // private field
  private int age;

  // getter method
  public int getAge() {
    return age;
  }

  // setter method
  public void setAge(int age) {
    this.age = age;
  }
}

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

    // create an object of Person
    Person p1 = new Person();

    // change age using setter
    p1.setAge(24);

    // access age using getter
    System.out.println("My age is " + p1.getAge()); /// My age is 24
  }
}
```

In the above example, we have a `private` field age. Since it is `private`, it cannot be accessed from outside the class.

In order to access age, we have used `public` methods: `getAge()` and `setAge()`. These methods are called getter and setter methods.

Making age private allowed us to restrict unauthorized access from outside the class. This is **data hiding**.

Abstract Classes

- **Cannot be instantiated using the new operator**
- **Usually contain abstract methods** that are implemented in concrete subclasses
  - E.g. computeArea() in GeometricObject
- Abstract classes and abstract methods are **denoted using the abstract modifier in the header**
- **A class that contains abstract methods must be defined as abstract**
  - **Abstract class can contain no abstract method**
- **If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined as abstract**

The **abstract class in Java cannot be instantiated** (we cannot create objects of abstract classes). We use the `abstract` keyword to declare an abstract class. For example,

```
// create an abstract class
abstract class Language {
  // fields and methods
}
...

// try to create an object Language
// throws an error
Language obj = new Language();
```

An abstract class can have both the regular methods and abstract methods. For example,

```
abstract class Language {

  // abstract method
  abstract void method1();
//In each of subclasses we implement this method

  // regular method
  void method2() {
    System.out.println("This is regular method");
  }
}
```

Implementing Abstract Methods

If the abstract class includes any abstract method, then all the child classes inherited from the abstract superclass must provide the implementation of the abstract method. For example,

```
abstract class Animal {
  abstract void makeSound();   <- Abstract method

  public void eat() {
    System.out.println("I can eat.");
  }
}

class Dog extends Animal {

  // provide implementation of abstract method
  public void makeSound() {
    System.out.println("Bark bark");
  }
}

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

    // create an object of Dog class
    Dog d1 = new Dog();

    d1.makeSound();
    d1.eat();
  }
}
```

Output

```
Bark bark
I can eat.
```

In the above example, we have created an abstract class Animal. The class contains an abstract method `makeSound()` and a non-abstract method `eat()`.

We have inherited a subclass Dog from the superclass Animal. Here, the subclass Dog provides the implementation for the abstract method `makeSound()`.

We then used the object d1 of the Dog class to call methods `makeSound()` and `eat()`.

Accesses Constructor of Abstract Classes

An abstract class can have constructors like the regular class. And, we can access the constructor of an abstract class from the subclass using the `super` keyword. For example,

```
abstract class Animal {
   Animal() {
      ….
   }
}

class Dog extends Animal {
   Dog() {
      super();
      ...
   }
}
```

Here, we have used the `super()` inside the constructor of Dog to access the constructor of the Animal.

Note that the `super` should always be the first statement of the subclass constructor.

```
abstract class MotorBike {
  abstract void brake();
}

class SportsBike extends MotorBike {

  // implementation of abstract method
  public void brake() {
    System.out.println("SportsBike Brake");
  }
}

class MountainBike extends MotorBike {

  // implementation of abstract method
  public void brake() {
    System.out.println("MountainBike Brake");
  }
}

class Main {
  public static void main(String[] args) {
    MountainBike m1 = new MountainBike();
    m1.brake();
    SportsBike s1 = new SportsBike();
    s1.brake();
  }
}
```

Key Points to Remember

- We use the `abstract` keyword to create abstract classes and methods.
- It **can have constructors and static methods also.**
- It **can have final methods** which will force the subclass not to change the body of the method.
- An abstract method doesn't have any implementation (method body).
- A class containing abstract methods should also be abstract.
- We **cannot create objects** of an abstract class.
- To implement features of an abstract class, we inherit subclasses from it and create objects of the subclass.
- A **subclass must override all abstract methods** of an abstract class. However, **if the subclass is declared abstract, it's not mandatory to override abstract methods.**
- We **can access the static attributes and methods of an abstract class using the reference of the abstract class.** For example,

  ```
  Animal.staticMethod();
  ```

Interfaces

An interface is a fully abstract class. It includes a group of abstract methods (methods without a body).

an interface is different from a class in several ways, including −

- You **cannot instantiate** an interface.
- An interface does **not contain any constructors.**
- All of the **methods** in an interface **are abstract.**
- An **interface cannot contain instance fields.** The **only fields that can appear in an interface must be declared both static and final.**
- An interface is **not extended by a class**; it is implemented by a class.
- An interface **can extend multiple interfaces**.

We use the `interface` keyword to create an interface in Java. For example,

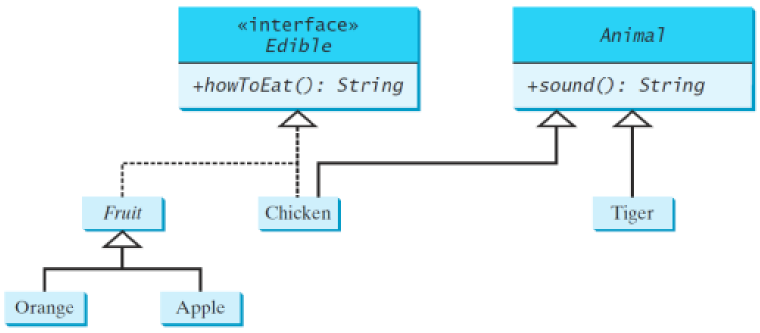
```
interface Language {
  public void getType();

  public void getVersion();
}
```

Here,

- Language is an interface.
- It includes abstract methods: `getType()` and `getVersion()`.

```
// create an interface
interface Language {
  void getName(String name);
}

// class implements interface
class ProgrammingLanguage implements Language {

  // implementation of abstract method
  public void getName(String name) {
    System.out.println("Programming Language: " + name);
  }
}

class Main {
  public static void main(String[] args) {
    ProgrammingLanguage language = new ProgrammingLanguage();
    language.getName("Java");
  }
}
```

- An interface can be used to define common behaviour for classes (including unrelated classes)
- **Contains only constants and abstract methods**
- Interfaces are denoted using the interface modifier in the header
- public interface Edible{
      public abstract String howToEat();
  }



UML (Unified Modelling Language) Model

```
abstract class Animal{
  public abstract String sound();
}

class Chicken extends Animal implements Edible{
  @Override
  public String howToEat(){
    return "Chicken: Fry it";
  }
  @Override
  public String sound(){
    return "Chicken: cock-a-doodle-doo";
  }
}
```

```
abstract class Fruit implements Edible{
  //Datafields, constructor, and metgods omitted here
}

class Apple extends Fruit {
  @Override
  public String howToEat(){
    return "Apple: Make apple cidar";
  }
}

class Orange extends Fruit {
  @Override
  public String howToEat(){
    return "Orange: Make orange juice";
  }
}
```

```
class Tiger extends Animal{
  @Override
  public String sound(){
    return "Tiger:
";
  }
}
```

Implementing Multiple Interfaces

In Java, a class can also implement multiple interfaces. For example,

```
interface A {
  // members of A
}

interface B {
  // members of B
}

class C implements A, B {
  // abstract members of A
  // abstract members of B
}
```

Extending an Interface

Similar to classes, interfaces can extend other interfaces. The `extends` keyword is used for extending interfaces. For example,

```
interface Line {
  // members of Line interface
}

// extending interface
interface Polygon extends Line {
  // members of Polygon interface
  // members of Line interface
}
```

Here, the Polygon interface extends the Line interface. Now, if any class implements Polygon, it should provide implementations for all the abstract methods of both Line and Polygon.

Extending Multiple Interfaces

An interface can extend multiple interfaces. For example,

```
interface A {
   ...
}
interface B {
   ...
}

interface C extends A, B {
   ...
}
```

Advantages of Interface in Java

Now that we know what interfaces are, let's learn about why interfaces are used in Java.

- Similar to abstract classes, interfaces help us to achieve **abstraction in Java**.

  Here, we know `getArea()` calculates the area of polygons but the way area is calculated is different for different polygons. Hence, the implementation of `getArea()` is independent of one another.

- Interfaces **provide specifications** that a class (which implements it) must follow.

  In our previous example, we have used `getArea()` as a specification inside the interface `Polygon`. This is like setting a rule that we should be able to get the area of every polygon.

  Now any class that implements the `Polygon` interface must provide an implementation for the `getArea()` method.

- **Interfaces are also used to achieve multiple inheritance in Java**. For example, Here, the class Rectangle is implementing two different interfaces. This is how we achieve multiple inheritance in Java.

  ```
  interface Line {
  …
  }

  interface Polygon {
  …
  }

  class Rectangle implements Line, Polygon {
  …
  }
  ```

  Rectangle

Note: All the methods inside an interface are implicitly `public` and all fields are implicitly `public static final`. For example,

```
interface Language {

  // by default public static final
  String type = "programming language";
  final int = 2;

  // by default public
  void getName();   // Doesnt matter if u use keyword abstract here or not
}
```

Generics

- Enable Type Parameterization
  - Generic Interfaces
  - Generic Classes
  - Generic Methods
- Example: **ArrayList** class
  - ArrayList<Integer> A = new ArrayList<Integer>();
  - ArrayList<String> B = new ArrayList<String>();
- **Generics types must be reference types**
- Generic types **could be bounded using the extends keyword**

The Java Generics allows us to create a single class, interface, and method that can be used with different types of data (objects).

This helps us to reuse our code.

Note: **Generics** does not work with primitive types (`int`, `float`, `char`, etc).

Java Generics Class

We can create a class that can be used with any type of data. Such a class is known as Generics Class.

Here's is how we can create a generics class in Java:

Example: Create a Generics Class

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

    // initialize generic class
    // with Integer data
    GenericsClass<Integer> intObj = new GenericsClass<>(5);
    System.out.println("Generic Class returns: " + intObj.getData());

    // initialize generic class
    // with String data
    GenericsClass<String> stringObj = new GenericsClass<>("Java Programming");
    System.out.println("Generic Class returns: " + stringObj.getData());
  }
}

// create a generics class
class GenericsClass<T> {

  // variable of T type
  private T data;

  public GenericsClass(T data) {
    this.data = data;
  }
  // method that return T type variable
  public T getData() {
    return this.data;
  }
}
```

Output

```
Generic Class returns: 5
Generic Class returns: Java Programming
```

In the above example, we have created a generic class named GenericsClass. This class can be used to work with any type of data.

```
class GenericsClass<T> {...}
```

Here, T used inside the angle bracket `<>` indicates the **type parameter**. Inside the `Main` class, we have created two objects of GenericsClass

- **intObj - Here, the type parameter T is replaced by `Integer`. Now, the GenericsClass works with integer data.**
- **stringObj - Here, the type parameter T is replaced by `String`. Now, the GenericsClass works with string data.**

Java Generics Method

Similar to the generics class, we can also create a method that can be used with any type of data. Such a class is known as Generics Method.

Here's is how we can create a generics class in Java:

Example: Create a Generics Method

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

    // initialize the class with Integer data
    DemoClass demo = new DemoClass();

    // generics method working with String
    demo.<String>genericsMethod("Java Programming");

    // generics method working with integer
    demo.<Integer>genericsMethod(25);
  }
}

class DemoClass {

  // create a generics method
  public <T> void genericsMethod(T data) {
    System.out.println("Generics Method:");
    System.out.println("Data Passed: " + data);
  }
}
```

Output

```
Generics Method:
Data Passed: Java Programming
Generics Method:
Data Passed: 25
```

In the above example, we have created a generic method named genericsMethod.

```
public <T> void genericMethod(T data) {...}
```

Here, the type parameter `<T>` is inserted after the modifier `public` and before the return type `void`.

We can call the generics method by placing the actual type `<String>` and `<Integer>` inside the bracket before the method name.

Note: We can **call the generics method without including the type parameter**. For example,

```
demo.genericsMethod("Java Programming");
```

In this case, the compiler can match the type parameter based on the value passed to the method.

Bounded Types

In general, the **type parameter** can accept any data types (except primitive types).

However, if we want to use **generics for some specific types (such as accept data of number types) only,** then we can use **bounded types.**

In the case of bound types, we use the `extends` keyword. For example,

```
<T extends A>
```

This means T can only accept data that are subtypes of A.

Example: Bounded Types

```
class GenericsClass <T extends Number> {

  public void display() {
    System.out.println("This is a bounded type generics class.");
  }
}

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

    // create an object of GenericsClass
    GenericsClass<String> obj = new GenericsClass<>(); //Error
  }
}
```

In the above example, we have created a class named GenericsClass. Notice the expression, notice the expression

```
<T extends Number>
```

Here, GenericsClass is created with bounded type. This means GenericsClass can only work with data types that are children of `Number` (`Integer`, `Double`, and so on).

However, we have created an object of the generics class with `String`. In this case, we will get the following error.

```
GenericsClass<String> obj = new GenericsClass<>();
                                                 ^
```

```
    reason: inference variable T has incompatible bounds
      equality constraints: String
      lower bounds: Number
  where T is a type-variable:
    T extends Number declared in class GenericsClass
```

Advantages of Java Generics

1. Code Reusability

With the help of generics in Java, **we can write code that will work with different types of data**. For example,

```
public <T> void genericsMethod(T data) {...}
```

Here, we have created a generics method. This same method can be used to perform operations on integer data, string data, and so on.

2. Compile-time Type Checking

The **type parameter** of generics provides information about the type of data used in the generics code. For example,

```
// using Generics
GenericsClass<Integer> list = new GenericsClass<>();
```

Here, we know that GenericsClass is working with `Integer` data only.

Now, if we try to pass data other than Integer to this class, the program will generate an error at compile time.

3. Used with Collections

The collections framework uses the concept of generics in Java. For example,

```
// creating a string type ArrayList
ArrayList<String> list1 = new ArrayList<>();

// creating a integer type ArrayList
ArrayList<Integer> list2 = new ArrayList<>();
```

In the above example, we have used the same ArrayList class to work with different types of data.

Similar to `ArrayList`, other collections (`LinkedList`, `Queue`, `Maps`, and so on) are also generic in Java.

Generics (The Comparable Interface)

- **Comparable** is a generic interface
  - Defines the **compareTo** method for comparing objects
- Defined as follows:

  ```
  public interface Comparable<T>{
      public int compareTo(T t);
  }
  ```

- The **compareTo** method determines the order of the calling object with **t** and returns a negative integer zero, or a positive integer if the calling object is less than, equal to. or greater than **t**
- Many classes implement Comparable (eg String Integer)

- Implementing comparable :

```
public class Point implements Comparable<Point> {
    // class body
    // blah blah

    @Override
    public int compareTo(Point p){
        // implementation
    }

}
```

Generics (The ArrayList class)

- Arrays cant be usred to store lists oif objects. Howevere, once an array is created, its size is fixed
- Java provides the generic class **ArrayList** whose **size is variable**
- imported using: import java.util.ArrayList
- Commonly used methods (ArrayList<E>)
    - boolean add(E e)
    - E get (int index)
    - int size()
    - boolean contains(Object o)
    - int indexOf(Object o)
- An ArrayList could be traversed using a for-each loop

For more info go to Lecture→ CSCB07 → collection framework

Generics (The HashSet class)

- Generic class that can be used to store elements **without duplicates**
    - No two elements e1 and e2 can be in the set such that e1.equals(e2) is true
- Imported using: import *java.util.HashSet;*
- **Objects added to the hash set should override *equals* and *hashCode* properly**
- Commonly used methods (HashSet)
    - boolean add(E e)
    - int size()
    - boolean contains(Object o)
- A HashSet could be traversed using a for-each loop

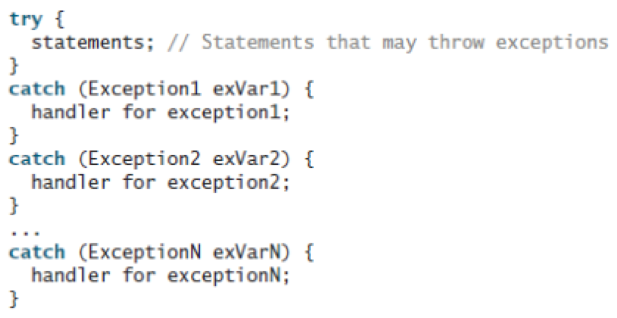
For more info go to Lecture→ CSCB07 → collection framework

Generics (The LinkedHashSet class)

- Elements of a HashSet are not necessarily stored in the same order they were added
- LinkedHashSet is a subclass of HashSet with a linked-list implementation that **supports an ordering of the elements** in the set
- Imported using: import java.util.LinkedHashSet;

Exceptions

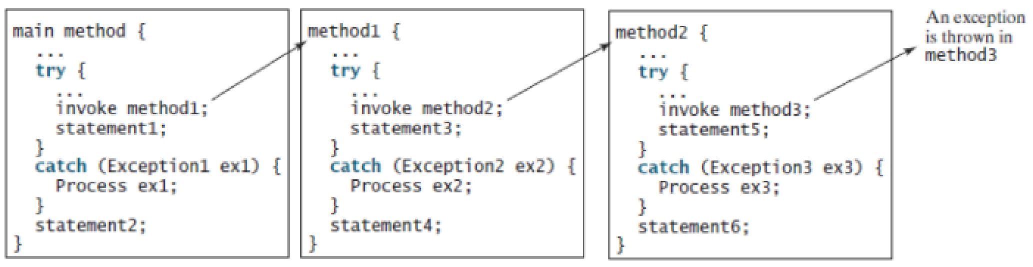
- Exception handling enables a **program to deal with exceptional situations and continue its normal execution**
- An **exception** is an object that represents an error or a condition that prevents execution from proceeding normally
- Exceptions are represented in the **Exception class, which describes errors caused by the program and by external circumstances**
- **Developers can create their own exception classes by extending Exception or a subclass of Exception**
- In Java, **runtime exceptions are represented in the RuntimeException** class. Subclasses include:
  - ArrayIndexOutOfBoundsException
  - NullPointerException
- **RuntimeException and its subclasses are known as** unchecked exceptions
- **All other exceptions are known as** checked exceptions
  - The compiler forces the programmer to check and deal with them in a trycatch block or declare them in the method header
- Declaring exceptions
  - **Every method must state the types of checked exceptions it might throw using the throws keyword in the header**
  - E.g. **public void myMethod() throws Exception1, Exception2, ..., ExceptionN**
- Throwing exceptions
  - A program that detects an error can create an instance of an appropriate exception type and throw it using the throw keyword
  - E.g. **throw new IllegalArgumentException("Wrong Argument");**
- When an exception is thrown, it can be caught and handled in a trycatch block. For example:



```
try {
  statements; // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVarN) {
  handler for exceptionN;
}
```

- If no exceptions arise during the execution of the try block, the catch blocks are skipped
- If **one of the statements inside the try block throws an exception**
  - The **remaining statements in the try block are skipped**

- **Each catch block is examined in turn, from first to last,** to see whether the type of the exception object is an instance of the exception class in the catch block
- **If no handler is found, Java exits this method, passes the exception to the method that invoked the method, and continues the same process to find a handler**
- **If no handler is found in the chain of methods** being invoked, the **program terminates** and **prints an error message** on the console



```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    Process ex1;
  }
  statement2;
}
```

```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    Process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    Process ex3;
  }
  statement6;
}
```

- **Java has a finally clause that can be used to execute some code regardless of whether an exception occurs or is caught.** For example:

```
try {
  //statements;
}
catch Exception ex) {
  //handling ex;
}
finally {
  //final statements;
}
```

Custom exception

```
class Lecture4Exception extends Exception{
  String message;
  public Lecture4Exception(String message){
    this.message = message
  }
}
```

```
public static void foo(int n) throws lecture4Exception{
  if (n< 0){
    throw new Lecture4Exception("Negative values are not
  }
  else{
    sysout("...");
  }
}
```

```
public class driver{

  //public static void bar(int n) throws Lecture4Exception{
  //    Person.foo(0);
  //}

//OR

  public static void bar(int n) {
    try{
      Person.foo(0);
      // someother exception then it goes to last exception
      sysout("!!!!!");   // Will be skipped if above throws exception
    }catch(Lecture4Exception ex){
        sysout("...")
    }catch(Exception ex){
        sysout(".../")
    }

  public static void main(String[] args){
    bar(0);
  }
}
```