GUJARAT TECHNOLOGICAL UNIVERSITY

MASTER OF COMPUTER APPLICATIONS

SEMESTER: III

Subject : **4639302**

Programming in JAVA

UNIT - 4

- Exception Handling,
- Generic Programming

Syllabus

- **Exception Handling:**
  - dealing with errors,
  - catching exceptions,
  - tips for using exceptions:
- **Generic Programming:**
  - A Simple Generic class,
  - generic methods,
  - bounds for type variables,
  - generic code and the VM,
  - restrictions and limitations and
  - inheritance rules for generic types.

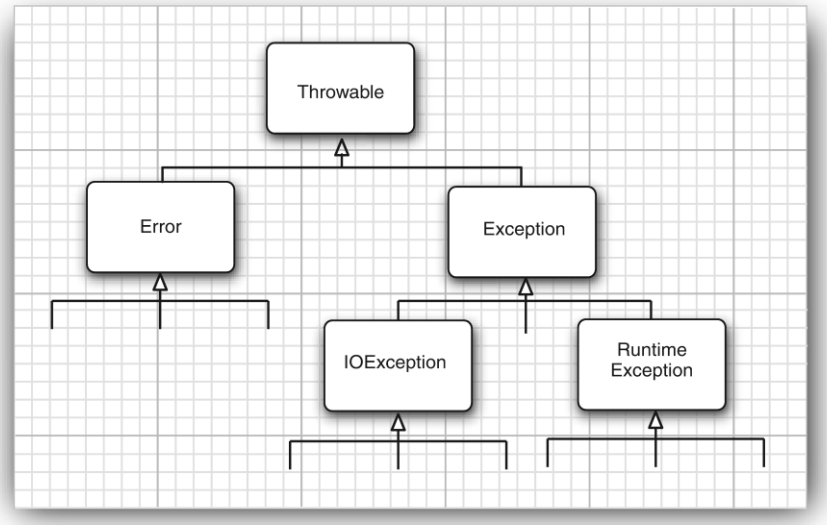
What is Exception Handling ?

- **Dictionary Meaning:**
  - Exception is an abnormal condition
- **JAVA:**
  - The **Exception Handling** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

Dealing with Errors

- **When an error occurs, your program can:**
  - Return to a safe state and allow the user to execute other commands.
  - Save the user's work and terminate.
- **What kind of errors do you need to consider?**
  - User input errors.
  - Device errors and physical limitations.
  - Code errors.
- **What can you do when an error occurs?**
  - Return an error code.
  - Terminate the program.
  - Throw an exception.

The Classification of Exceptions



- Errors are internal problems:
  - exhaustion of memory or other resources.
- Runtime exceptions are the programmer's fault:
  - null pointers, out of bounds values, and so on.
- Other exceptions are unavoidable facts of life:
  - flaky network connections, files that were just moved, and so on.

Types of Java Exceptions

1. Checked Exception

- The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions
- **Example:** IOException, SQLException etc.
- Checked exceptions are checked at compile-time.

2. Unchecked Exception

- The classes which inherit RuntimeException are known as unchecked exceptions
- **Example:** ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.
- Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3. Error

- Error is irrecoverable
- **Example:** OutOfMemoryError, VirtualMachineError, AssertionError etc.

Declaring Checked Exceptions

- The “facts of life” exceptions are checked exceptions: The compiler checks that you deal with them.
- These are subclasses of Exception but not of RuntimeException.
- If you write a method that might throw such an exception, you need to declare that fact.
- Add a throws clause:
  - **public Image loadImage(String s) throws IOException**
- A throws clause can list multiple exceptions:
  - **public Image loadImage(String s) throws FileNotFoundException, EOFException**
- Don't declare unchecked exceptions:
  - **void drawImage(int i) throws ArrayIndexOutOfBoundsException // bad style**
  - Instead, fix your code so that this doesn't happen!

Example of unchecked exceptions

- **ArithmeticException**

  int a=50/0;//ArithmeticException

- **NullPointerException**

  String s=null;

  System.out.println(s.length());//NullPointerException

- **NumberFormatException**

  String s="abc";

  int i=Integer.parseInt(s);//NumberFormatException

- **ArrayIndexOutOfBoundsException**

  int a[]=new int[5];

  a[10]=50; //ArrayIndexOutOfBoundsException

Keywords in Exception

Keyword	Description
try	• It is used to specify a block where we should place exception code. • The try block must be followed by either catch or finally. • It means, we can't use try block alone.
catch	• It is used to handle the exception. • It must be preceded by try block which means we can't use catch block alone. • It can be followed by finally block later.
finally	• It is used to execute the important code of the program. • It is executed whether an exception is handled or not.
throw	• It is used to throw an exception.
throws	• It is used to declare exceptions. • It doesn't throw an exception. • It specifies that there may occur an exception in the method. • It is always used with method signature.

Catching an Exception

- Java try block is used to enclose the code that might throw an exception.
- Java try block must be followed by either catch or finally block.

```
try
{
  //code that may throw exception
}
catch(ExceptionType e)
{
}
```

```
try
{
//code that may throw exception
}
finally
{
}
```

- If an exception is thrown, and nobody catches it, your program terminates.

Example

```
public class X
{
   public static void main(String args[])
   {
        int data=100/0;  //error
        System.out.println("after exception");
    }
}
```

Output:

Exception in thread main java.lang.ArithmeticException:/ by zero

```
public class X
{
   public static void main(String args[])   {
        try
        {
                int data=100/0;  //error
        }
        catch(ArithmeticException e)  {
                System.out.println(e);
        }
        System.out.println("after exception");
    }
}
```

Output:

Exception in thread main java.lang.ArithmeticException:/ by zero

after exception

What JVM does when Exception occurs?

- The JVM firstly checks whether the exception is handled or not.
- **If exception is not handled**, JVM provides a default exception handler that performs the following tasks:
  - Prints out exception description.
  - Prints the stack trace (Hierarchy of methods where the exception occurred).
  - Causes the program to terminate.
- But if **exception is handled** by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

Catching Multiple Exceptions

```
try
{
   code that might throw exceptions
}
catch (FileNotFoundException | UnknownHostException e)
{
   emergency action for missing files and unknown hosts
}
catch (IOException e)
{
   emergency action for all other I/O problems
}
```

- **Rules:**
  - **At a time only one Exception is occurred and only one catch block is executed.**
  - **All catch blocks must be ordered from most specific to most general**

Catching Multiple Exceptions
Inheritance hierarchy of exceptions:

```
class A {
    public static void main(String args[])
   {
            try{
               int a[]=new int[5];
               a[5]=30/0; // error
            }
            catch(ArithmeticException e)   {System.out.println("e1");}
            catch(ArrayIndexOutOfBoundsException e)   {System.out.println("e2");}
            catch(Exception e)   {System.out.println("e3");}
            System.out.println("after exception");
   }
}
```

Output:

```
e1
after exception
```

Work with the inheritance hierarchy of exceptions: Catch more specific exceptions before more general ones.

Catching Multiple Exceptions

Inheritance hierarchy of exceptions:

```
class A {
     public static void main(String args[])
    {
            try{
               int a[]=new int[5];
               a[5]=30/0; // error
            }
            catch(Exception e)   {System.out.println("e3");}
            catch(ArithmeticException e)   {System.out.println("e1");}
            catch(ArrayIndexOutOfBoundsException e)   {System.out.println("e2");}
           System.out.println("after exception");
    }
}
```

Output:

Compile - time error

finally block

- Java finally block is always executed whether exception is handled or not.
- Java finally block follows try or catch block.
- It is used *to execute important code* such as closing connection, stream etc.

```
try
{
//code that may throw exception
}
catch(Exception e)
{
}
finally
{
}
```

- For each try block there can be zero or more catch blocks, but only one finally block.
- The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

throw exception

- The Java throw keyword is used to explicitly throw an exception.
- We can throw either checked or uncheked exception using throw keyword.
- The throw keyword is mainly used to throw custom exception.

Syntax: **throw** exception;

```
public class ThrowDemo
{
    static void validate(int age)
    {
       if(age<18)
         throw new ArithmeticException("not valid");
       else
         System.out.println("welcome to vote");
    }
    public static void main(String args[])
    {
        validate(13);
        System.out.println("rest of the code...");
    }
}
```

Exception in thread main java.lang.ArithmeticException:not valid

Custom / user-defined Exception

- If you are creating your own Exception that is known as custom exception or user-defined exception.
- Java custom exceptions are used to customize the exception according to user need.
- By the help of custom exception, you can have your own exception and message.

create custom Exception class

Extends your class from Exception

```
class InvalidAgeException extends Exception
 {
        InvalidAgeException(String s)
        {
             super(s);
        }
}
```

throw custom Exception class

```
class CustomExceptionDemo
{
  static void validate(int age) throws InvalidAgeException
  {
    if(age<18)
                throw new InvalidAgeException("not valid");
    else
                System.out.println("welcome to vote");
  }
  public static void main(String args[])
  {
        try {                validate(13);      }
        catch(Exception m)   {
                System.out.println("Exception occured: "+m);   }
        System.out.println("rest of the code...");
  }
}
```

Rethrowing and Chaining Exceptions

- **Sometimes you want to catch an exception and rethrow it as a different type:**

```
try
{
        access the database
}
catch (SQLException e)
{
        throw new ServletException("database error: " +
                                   e.getMessage());
}
```

throws keyword

- The **Java throws keyword** is used to declare an exception.
- It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.
- Exception Handling is mainly used to handle the checked exceptions.
- If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

Syntax of throws

```
returnType methodName() throws exceptionClassName
{
        //method code
}
```

- *Which exception should be declared*
  - only **checked exception** is declared:
  - **unchecked Exception:**
    - under our control so we should correct our code.
  - **error:**
    - beyond our control
    - e.g. we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Difference between

throw and throws

throw	throws
Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
Throw is followed by an instance.	Throws is followed by class.
Throw is used within the method.	Throws is used with the method signature.
You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Example

Example: throw

```
void myMethod()
{
        throw new ArithmeticException("sorry");
}
```

Example: throws

```
void myMethod() throws ArithmeticException
{
        //method code
}
```

Tips for Using Exceptions

- Exception handling is not supposed to replace a simple test.

```
try
{
   s.pop();
}
catch (EmptyStackException e)
{
}
```

⇒

```
if (!s.empty()) s.pop();
```

- Do not micromanage exceptions.

```
for (i = 0; i < 100; i++)
{
   try { n = s.pop(); }
   catch (EmptyStackException e) { . . . }
   try { out.writeInt(n); }
   catch (IOException e) { . . . }
}
```

⇒

```
try
{
   for (i = 0; i < 100; i++)
   {
      n = s.pop();
      out.writeInt(n);
   }
}
catch (IOException e) { . . . }
catch (EmptyStackException e) { . . . }
```

Tips for Using Exceptions

- Make good use of the exception hierarchy:
  - Don't just throw a RuntimeException. Find an appropriate subclass or create your own.
  - Don't just catch Throwable.
  - Respect the difference between checked and unchecked exceptions.
  - Do not hesitate to turn an exception into another exception that is more appropriate.
- Do not squelch exceptions:

```
try
{
        code that threatens to throw checked exceptions
}
catch (Exception e)
{  } // so there
```

Tips for Using Exceptions

- When you detect an error, do better than just tolerance.
  - When something is very wrong, throw an exception.
  - Don't return an error code or a dummy value.
  - Return values must be handled by the caller. Exceptions can be handled *anywhere* upstream.
- Propagating exceptions is not a sign of shame.
  - Don't try to handle an exception that you can't cure.
  - Just let it be rethrown so that it can reach a competent handler.
- These two rules can be summarized as: “throw early, catch late.”

Generic Programming

Why Generic Programming?

- The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects.
- Before generics, we can store any type of objects in collection i.e. non-generic.
- Now generics, forces the java programmer to store specific type of objects.

Advantage of Java Generics

1) Type-safety :

- We can hold only a single type of objects in generics.
- It doesn't allow to store other objects.

2) Type casting is not required:

- There is no need to typecast the object.

3) Compile-Time Checking:

- It is checked at compile time so problem will not occur at runtime.
- The good programming strategy says it is far better to handle the problem at compile time than runtime.

2) Type casting is not required:

Before Generics, we need to type cast.

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
```

After Generics, we don't need to typecast the object.

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

3) Compile-Time Checking:

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);//Compile Time Error
```

Who Wants to Be a Generic Programmer?

- Generic classes such as ArrayList are easy to use.
- It is not so easy to implement ArrayList.
  - The class has to work for all element types.
  - Should it be legal to call addAll to add an ArrayList<Manager> to an ArrayList<Employee>?
  - If so, how do you allow that call and disallow the opposite?
- **Three skill levels:**
  - Use existing generic classes.
  - Understand enough about type parameters to resolve problems.
  - Write generic classes that others can use.

Define a Simple Generic Class

```
public class Pair<T>
{
        private T first;
        private T second;

        public Pair() { first = null; second = null; }
        public Pair(T first, T second) { this.first = first; this.second = second; }

        public T getFirst() { return first; }
        public T getSecond() { return second; }

        public void setFirst(T newValue) { first = newValue; }
        public void setSecond(T newValue) { second = newValue; }
}
```

Type Variables

- The T in public class Pair<T> is a type variable.
- The type variable is used throughout the class definition:
  - private T first;
- Instantiate the type like Pair<String>, substituting a type for the variable.
- You can think of the result as an ordinary class with these methods:
  - String getFirst()
  - String getSecond()
  - void setFirst(String)
  - void setSecond(String)
- A class can have multiple type variables:

```
public class Pair<T, U>
{
  T first;
  U second;
  . . .
}
```

Generic Methods

- Generic class is a Class with type variables.
- Generic method is a Method with type variables:

```
class ArrayAlg
{
    public static <T> T getMiddle(T... a)
    {
       return a[a.length / 2];
    }
}
```

Generic Methods

- **When you call a generic method, the actual type comes before the method name:**

  String middle = ArrayAlg.<String>getMiddle("John", "Q.", "Public");

- **But you rarely have to do that. Usually, the compiler infers the type from the argument types:**

  String middle = ArrayAlg.getMiddle("John", "Q.", "Public");

- **Occasionally, something goes wrong:**

  double middle = ArrayAlg.getMiddle(3.14, 17.29, 0);

  // Type mismatch: cannot convert from Number&Comparable<?> to double

Bounds for Type Variables

- Sometimes, a type variable cannot be instantiated with arbitrary types:

```
class ArrayAlg
{
        public static <T> T min(T[] a) // almost correct
        {
                if (a == null || a.length == 0) return null;
                T smallest = a[0];
                for (int i = 1; i < a.length; i++)
                   if (smallest.compareTo(a[i]) > 0) smallest = a[i];
                return smallest;
        }
}
```

Bounds for Type Variables

- How do we know that T has a compareTo method?
- Need to restrict T in the method declaration:

  public static <T extends Comparable> T min(T[] a) ...
- Now min can be called with arrays of String, LocalDate, and so on, but not Rectangle.
- **Note:** Comparable also has a type parameter which we will ignore for a little longer.
- A type variable can have multiple bounds:

  T extends Comparable & Cloneable

Wildcard in Java Generics

- The ? (question mark) symbol represents wildcard element.
- It means any type.
- If we write <? extends Number>,
  - it means any child class of Number
  - e.g. Integer, Float, double etc.
- Now we can call the method of Number class through any child class object.

Example

//creating a method that accepts only child class of Shape

```
public static void drawShapes(List<? extends Shape> lists)
{
        for(Shape s:lists)
        {
                s.draw();
        }
}
```

Understand how generic code is translated to run on the Java virtual machine

Type Erasure

- The Java virtual machine has no notion of generic types or methods.
- Generic classes and methods turn into ordinary classes and methods.
- Type variables are “erased”, yielding a *raw type*:

```
public class Pair
{
        private Object first;
        private Object second;
        public Pair(Object first, Object second) { . . . }
        public Object getFirst() { return first; }
        public Object getSecond() { return second; }
        public void setFirst(Object newValue)
        { first = newValue; }
        public void setSecond(Object newValue)
        { second = newValue; }
}
```

Cast Insertion

- The compiler inserts casts when a return type has been erased:

```
Pair<Employee> buddies = . . .;
Employee buddy = buddies.getFirst();
```

⇒

```
Pair buddies = . . .;
Employee buddy = (Employee) buddies.getFirst();
```

- Casts are not needed for erased parameter types:
  - buddies.setFirst(buddy); // OK to convert Employee to Object
- Exception: Multiple bounds require casts to first bound.

Bridge Methods

- Consider this class:

```
class DateInterval extends Pair<LocalDate>
{
        public void setSecond(LocalDate second)
        {
           if (second.compareTo(getFirst()) >= 0)
              super.setSecond(second);
        }
        . . .
}
```

Bridge Methods

- **After erasure:**

  class DateInterval extends Pair

- **Two setSecond methods:**

  public void setSecond(LocalDate second) { . . . }

  public void setSecond(Object second)

  { setSecond((LocalDate) second); }

- **Second method is needed for polymorphism:**

  Pair<LocalDate> pair = new DateInterval();

  pair.setSecond(aDate);

Calling Legacy Code

- When generics were added to Java, a major goal was to interoperate with legacy code.
- Example: Legacy class Department with methods

  ArrayList getEmployees()

  void addAll(ArrayList employees)

- This call generates a warning:

```
ArrayList<Employee> newHires = . . .;
dept.addAll(newHires);
```

- Remedy: Annotate method with **@SuppressWarnings("unchecked")**
- In the opposite case, can annotate just the variable holding the returned value:

```
@SuppressWarnings("unchecked") ArrayList<Employee> result =
dept.getEmployees();
```

restrictions and limitations of Java generics

Primitive Types

- Since type variables are erased to Object, they don't work for primitive types.
  - Remedy 1: Use wrapper classes.
  - Remedy 2: Make added versions for primitive types.
  - Example: Consumer, IntConsumer, LongConsumer, DoubleConsumer

Runtime Type Inquiry

- **instanceof tests only work with raw types:**
  - if (a instanceof Pair<String>) // Error
- **Casts with parameterized types give a warning:**
  - Pair<String> p = (Pair<String>) a; // Warning--can only test that a is a Pair
- **The getClass method returns the raw type:**
  - Pair<String> stringPair = . . .;
  - Pair<Employee> employeePair = . . .;
  - if (stringPair.getClass() == employeePair.getClass()) // they are equal

Creating Arrays

- You cannot instantiate arrays of parameterized types:
  - Pair<String>[] pairs = new Pair<String>[10]; // Error
- Suppose it worked. After erasure:
  - Pair[] pairs = new Pair[10];
- An array remembers its component type so that you can't store an element of the wrong type:
  - Object[] objects = pairs;
  - objects[0] = "Fred"; // ArrayStoreException: Can't store a String into a Pair[]
- But the array only remembers its raw type:
  - objects[0] = new Pair<Integer>(...); // No ArrayStoreException
- **Since arrays of generic types are unsound, they are illegal.**

Instantiating Types

- **You cannot instantiate a generic type:**

  public Pair() { first = new T(); second = new T(); } // Error

- **Can remedy by making caller pass a constructor expression:**

  Pair<String> p = Pair.makePair(String::new);

- **Implementation:**

```
public static <T> Pair<T> makePair(Supplier<T> constr)
{
        return new Pair<>(constr.get(), constr.get());
}
```

- **Called like this:**

  Pair<String> p = Pair.makePair(String.class);

Static Contexts

- **You cannot use type variables in static fields or methods.**
- For example, this doesn't work:

```
public class Singleton<T>
{
        private static T singleInstance; // Error
        public static T getSingleInstance() // Error
        {
                if (singleInstance == null) construct new instance of T
                    return singleInstance;
        }
}
```

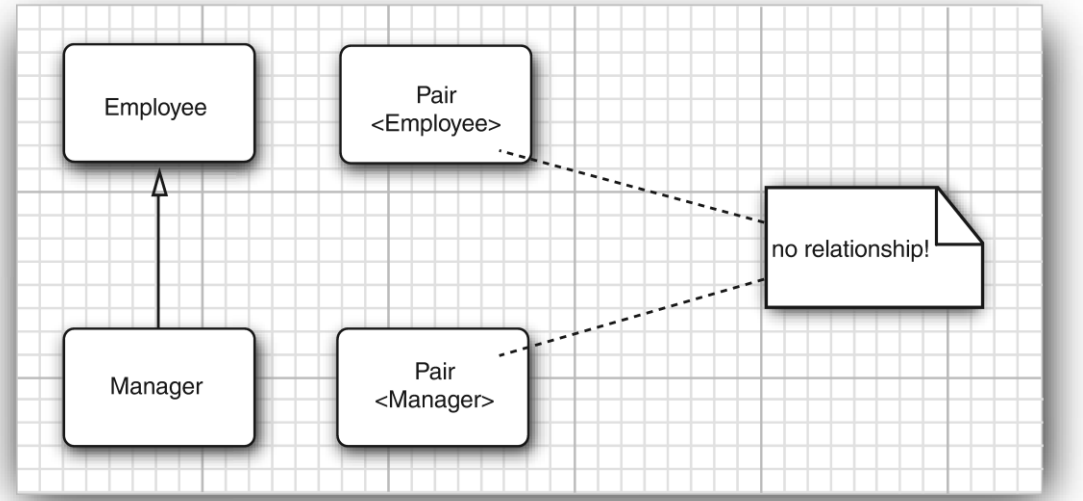
Exceptions

- You cannot throw or catch objects of generic type
- You cannot use a type variable in a catch clause

Understand the interaction between generic types and inheritance

Inheritance and Subtype Relationships

- Manager is a subclass of Employee.
- Is Pair<Manager> a subclass of Pair<Employee>?
- No subtype relationship between GenericType<Type1> and GenericType<Type2>.



Contact:

Bipin S. Rupadiya

Assistant Professor, JVIMS

Mo. : +91-9228582425

Email: info@bipinrupadiya.com

Blog : www.BipinRupadiya.com