Advanced Programming

Exceptions

What Is an Exception?

An "abnormal event" that occurs during the execution

```
public class Example {
  public static void main(String args[]) {
    int v[] = new int[10];
    v[10] = 0; //Oops... !
    System.out.println("Hello world ...?!");
  }
}
```

"Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException :10
at Example.main (Example.java:4)"

"throw an exception"

"catch the exception"

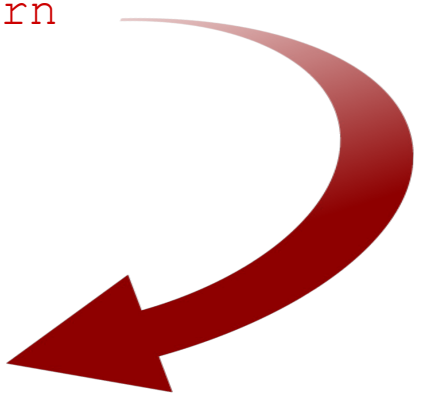
"exception handler"

Catching and Handling Exceptions

try - catch - finally

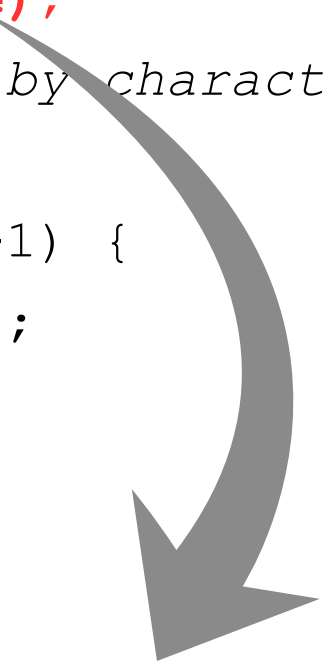
```
try {
  // Block of instructions
  methodX()
  methodY()
  methodZ()
}
catch (ExceptionType1 variable) {
  // Handling exceptions of type 1
}
catch (ExceptionType2 variable) {
  // Handling exceptions of type 2
}
catch (ExceptionType3 | ExceptionType4 variable) {
  // Handling exceptions of type 3 or 4
}
finally {
  // Cleanup code: executes whether or not an exception is thrown
}
...execution continues
```

A new exception is born



Example – Reading a File

```
public static void readFile(String filename) {
   FileReader f = null;
   // Open the file
   f = new FileReader(filename);
   // Read the file character by character
   int c;
   while ( (c = f.read()) != -1) {
      System.out.print((char)c);
   }
   // Close the file
   f.close();
}
```



unreported exception FileNotFoundException;
must be caught or declared to be thrown

“Catching” I/O Exceptions

```
public static void readFile(String filename) {
  FileReader f = null;
  try {
    f = new FileReader(filename);    // Open the file
    int c;                           // Read the file
    while ( (c=f.read()) != -1) {
      System.out.print((char)c);
    }
  } catch (FileNotFoundException e) {
    System.err.println("The file " + filename + "is missing!");
  } catch (IOException e) {
    System.out.println("Unexpected error reading the file!");
    e.printStackTrace();
  } finally {
    if (f != null) {                 // Close the file
      try {
        f.close();
      } catch (IOException e) {
        System.err.println("Error closing the file!");
      }
    }
  }
}
```

“Throwing” exceptions

```
[modifiers] ReturnedType method([arguments])
   throws ExceptionType1, ExceptionType2, … { }
```

```
public class FileReadExample {
  public static void readFile(String filename)
      throws FileNotFoundException, IOException {
    FileReader f = new FileReader(filename);
    int c;
    while ( (c = f.read()) != -1)
      System.out.print((char)c);
    f.close();
  }
  public static void main(String args[]) {
    try {
      readFile(args[0]);
    } catch (FileNotFoundException e) {
      System.err.println("File not found...");
    } catch (IOException e) {
      System.out.println("I/O Error");
    } catch (Exception e){
      System.out.println("Something is wrong..." + e);
    }
  }
}
```

try - finally

```
public static void readFile(String filename)
    throws FileNotFoundException, IOException {
  FileReader f = null;
  try {
    f = new FileReader(filename);
    int c;
    while ( (c=f.read()) != -1)
      System.out.print((char)c);
  }
  finally {
    if (f!=null)
      f.close();
  }
}
```

try-with-resources

Resource - objects that must be closed after using them; they are of type *AutoCloseable*

```
try(FileReader f = new FileReader(filename)) {
  int c;
  while ( (c = f.read()) != -1)
    System.out.print((char)c);
}
```

The exception thrown when the *FileReader* was closed is suppressed

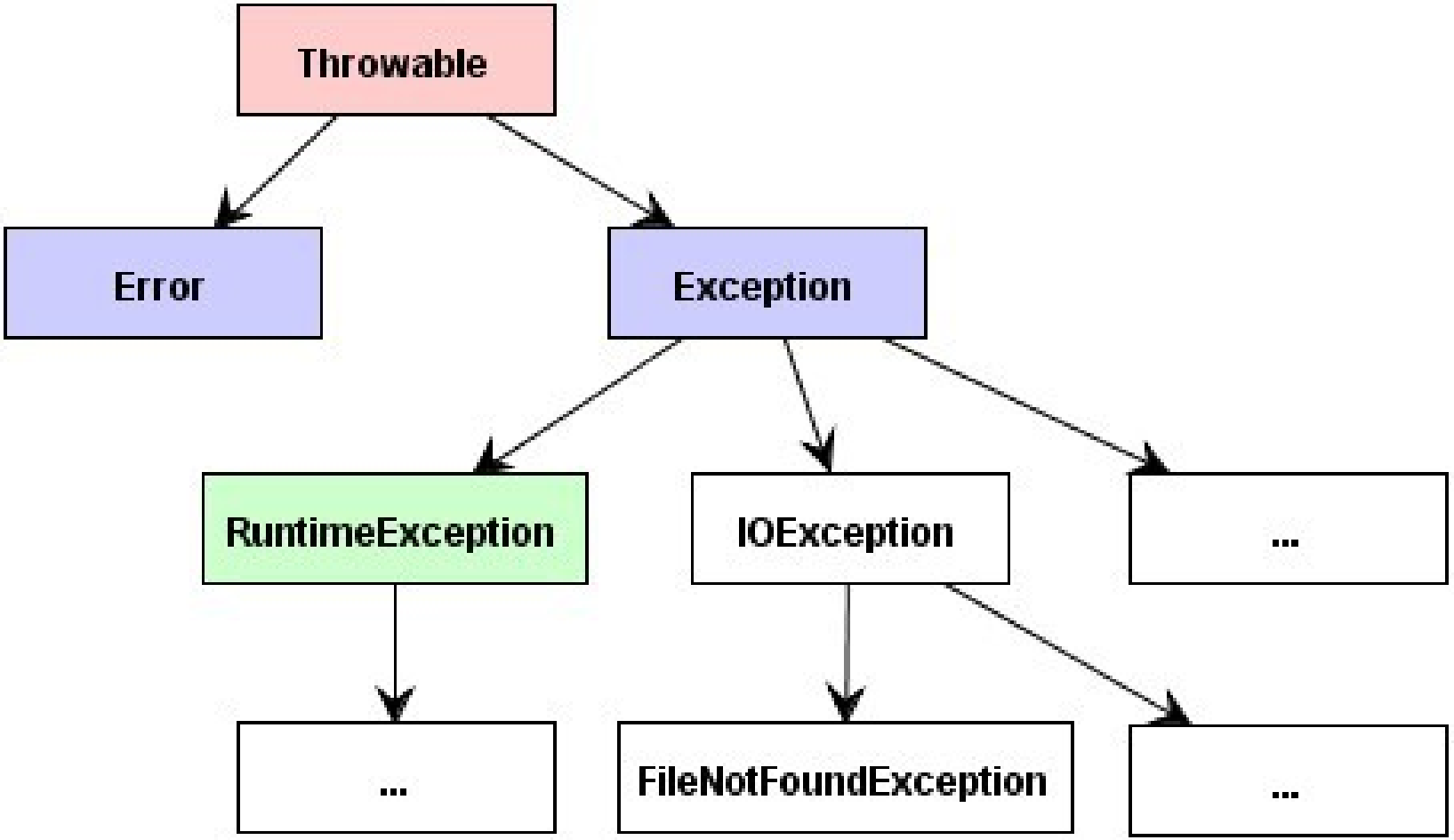
```
try (Connection con = createConnection();
     Statement stmt = con.createStatement();
     ResultSet rs = stmt.executeQuery(query)) {

  return (rs.next() ? rs.getObject(1) : null);
} catch (SQLException e) {
  System.err.println(e);
}
```

The resources will be closed in reverse order

The **close()** method of an **AutoCloseable** object is called automatically when exiting a try-with-resources block for which the object has been declared in the resource specification header.

Exceptions Class Hierarchy



Checked versus *Unchecked*

- *Checked Exceptions*
  - Abnormal situations that can not be controlled at the time of writing the code (design-time): *file system* errors, *network* communications errors, etc.
  - Must be handled ("caught" or "thrown")
  - Extend *Exception*: `IOException, SQLException, etc.`
- *Unchecked Exceptions*
  - Errors caused by situations out of which the application can not recover, usually *programming mistakes*.
  - Do not need to be handled (but it is possible)
  - Extend either *Error* or *RuntimeException*:
    `NullPointerEception, IllegalArgumentException, ArithmeticException, ArrayIndexOutOfBoundsException, etc.`

Error

- Indicates a **serious problem** that a reasonable application should not try to catch. Most such errors are abnormal conditions.
- *Unchecked*
- Examples:
  - **`VirtualMachineError`**

    (Thrown to indicate that the Java Virtual Machine is broken or has run out of resources necessary for it to continue operating.)

    - **`InternalError, UnknownError,`**
    - **`OutOfMemoryError, StackOverflowError`**

Who Creates the Exceptions?

The throw Statement

- The author of a method will signal exceptional situations **creating** and throwing exceptions.

```
public class Stack {
  …                          //throws is mandatory for checked exceptions
  public Object peek() throws EmptyStackException {
    int len = size();
    if (len == 0) throw new EmptyStackException();
    return elementAt(len - 1);
  }
}
```

- The Virtual Machine, for "standard" *RuntimeExceptions*

Validating Method Arguments

```
class Person {
  private String name;
  private int age;

  public void setName(String name) {
    if (name == null || name.trim().equals("")) {
        throw new IllegalArgumentException(
          "Name should not be empty.");
    }
    if (!name.matches("[a-zA-Z]+")) {
        throw new IllegalArgumentException(
          "Name should only contain characters: " + name);
    }
    this.name = name;
  }

  public void setAge(int age) {
    if (age < 0) {
      throw new IllegalArgumentException(
        "Age should be a positive number: " + age);
    }
    this.age = age;
  }
  ...
}
```

Reusing Validation Rules

- Verifying data integrity is a **repetitve work**
- Error messages may also **vary by locale**
- Consider creating your own "**validation framework**" or use an existing one, such as Apache Commons Validator (or other)
- Example: implement a class for validating emails:

```
public void setEmail(String email) {
  new EmailValidationRule().validate(email);
  ...
}
```

Creating Custom Exceptions

- Extend a proper subclass of *Throwable*
- *Checked* vs. *Unchecked*:

  If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

```
public class MyOwnException extends Exception {

  //Usually, define custom properties and constructors

  public MyOwnException(String message) {
    super(message);
  }
}
```

RuntimeException

Example

- Define your custom exception

```
public class InvalidAgeExeption extends RuntimeException {
    public InvalidAgeExeption(String message) {
        super(message);
    }
    public InvalidAgeExeption(int age) {
        super("Invalid age: " + age);
    }
}
```

- Use your custom exception

```
public void setAge(int age) {
  if (age < 0)  throw new InvalidAgeException(age);
  if (age > 18) throw new InvalidAgeException("Sorry, too old");
  this.age = age;
}
```

Exception Chaining (Wrapping)

```
try {
   Person person =
     database.readPerson(personId);

} catch (SQLException sqlException) {
    // catch the original exception
    System.err.println(sqlException);

    // create a new, custom exception
    // wrapping the original exception
    MyException myException =
      new MyException("Database Error", sqlException);
    myException.setDetail("Invalid person id " + personId);

    // throw the custom exception
    throw myException;
}
```

Code Separation

"Spaghetti" Code (tangled, unstructured)

```
int readFile() {
  int codEroare = 0;
  open the file;
  if (the file has opened) {
    determine its size;
    if (the size was determined ) {
      allocate memory;
      if (the memory was allocated) {
        read the file into memory;
        if (cannot read from file){
          errorCode = -1;
        }
      } else {
        errorCode = -2;
      }
    }
    ...
  }}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}
  return errorCode;
}
```

```
int readFile() {
try {
  open the file;
  determine its size;
  allocate memory;
  read the file into memory;
  close the file;

} catch(file didn't open){
  handle this exception
} catch (cannot determine size){
  handle this exception
} catch (not enough memory) {
  handle this exception
} catch (file read failed) {
  handle this exception
} catch (cannot close the file) {
  handle this exception
}
```

Grouping and Differentiating Error Types

```
try {

  String driverName = new String(
    Files.readAllBytes(Paths.get("driver.txt")));

  Class.forName(driverName).newInstance();

} catch (IOException ex) {
  // problems with the file

} catch (ClassNotFoundException ex) {
  // problem with the driver class

} catch (IllegalAccessException ex) {
  // cannot acces the class

} catch (InstantiationException ex) {
  // cannot instantiate the class
}
```

✔ Propagating Errors

```
int method1() {
  try {
    method2();
  } catch (ExceptionType e) {
    //handle the exception
  }
}
int method2() throws ExceptionType {
  ...
  method3();
  ...
}

int method3() throws ExceptionType {
  ...
  throw new ExceptionType();
  ...
}
```