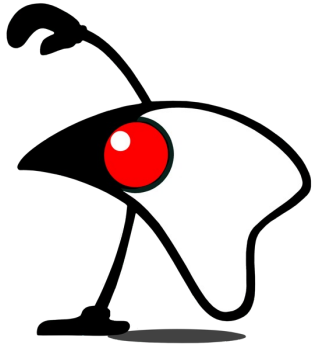
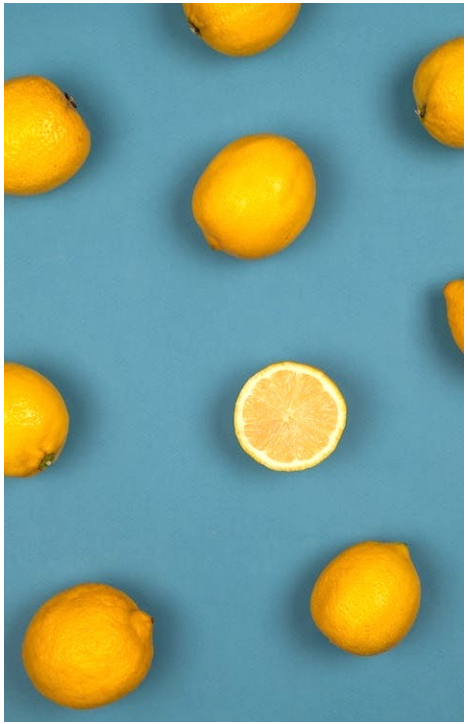
Exception Handling and Serialization (I/O)



CO7005 Software Development Techniques

Dr Stuart Cunningham
s.cunningham@chester.ac.uk

Exceptions



- Errors that occur in programs at runtime
  - Known as *throwing* an exception
- Often due to unforeseen conditions or events
- Resulting in undesirable behaviour or program termination
- Java has standard exceptions, so we know what to look out for
- *Exception objects*, contain information about what happened

Java Exceptions – Key Terminology

Keyword	Definition
`try`	Code block designed to be the primary path through the program. This code has the potential to cause (*throw*) an exception.
`catch`	Block of code that defines what to do if an exception occurs in the corresponding try block – defines how to *handle* the exception.
`finally`	Code to be executed when either the try or catch blocks complete. Use for *essential* code.
`throw`	Allows code to manually throw (generate) an exception and facilitates custom exceptions.
`throws`	Specifies exception(s) that *may* be thrown by a method.

Java Exceptions - Common Examples

- *Checked* exceptions (*require* handling to compile)
  - FileNotFoundException
  - IOException
  - ClassNotFoundException
- *Unchecked* exceptions (*don't* require handling to compile)
  - ArithmeticException
  - ArrayIndexOutOfBoundsException
  - NullPointerException

“Checked exceptions are intended to cover application-level problems, such as missing files and unavailable hosts. As good programmers (and upstanding citizens), we should design software to ***recover gracefully*** *from these kinds of conditions. Unchecked exceptions are intended for system-level problems …you don’t have to wrap every one of your array-index operations in a try/catch statement…”*

(Loy 2020)

Exception Handling

```
// see if this works
try
{
 int idx = Integer.parseInt(args[0]);
 int[] numbers = {1, 2, 3};
 System.out.println(numbers[idx]);
}

// do this if an exception occurs
catch (ArrayIndexOutOfBoundsException e)
{
 System.out.println("Index invalid");
 System.out.println(e);
}
```

- Java has an *exception handler*
  - Specifies action we want a program to *normally* perform
  - Provides *contingency* actions should an exception occur
- Using `try – catch` syntax
  - A bit like `if – else`
  - Must be paired
- `finally` gives additional control

```java
public class TryCatchFinally {
 public static void main(String[] args) {
  // see if this works
  try {
    // get idx value from terminal
    int idx = Integer.parseInt(args[0]);
    int[] numbers = {1, 2, 3};
    System.out.println(numbers[idx]);
  }

  // do this if an exception occurs
  catch(ArrayIndexOutOfBoundsException e) {
    System.out.println("Index invalid");
    System.out.println(e);
  }

  // do this anyway
  finally {
    System.out.println("Program Terminates");
  }
 }
}
```

```
>> java TryCatchFinally 5
Index invalid
java.lang.ArrayIndexOutOfBoundsException
: Index 5 out of bounds for length 3
Program Terminates

>> java TryCatchFinally 0
1
Program Terminates
```

Exception Handling Details

- If an exception occurs in a line, the `try` block ends there
- Control is transferred to the `catch` block
- The `catch` applies to any methods called in a `try` block
- Exception prevents unwanted program termination
- Once an exception is handled is it removed
- Multiple `catch` blocks can be paired with one `try` block
  - Can use *separate* or *combined* (with OR) exceptions statements

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

  int[] a = {10, 20, 40, 50};
  int[] b = {0, 2, 4, 5, 8};

  for (int i = 0; i <b.length; i++){
    try {
       System.out.println(a[i]/b[i]);
    }
    // catch Arithmetic problems
    catch (ArithmeticException ea) {
       System.out.println("Arithmetic Error ");
       System.out.println("\t"+ea);
    }
    // catch Array Index problems
    catch (ArrayIndexOutOfBoundsException ei) {
       System.out.println("Array Index Error ");
       System.out.println("\t"+ei);
    }
  }
}
```

```
>> java MultiCatch
Arithmetic Error
    java.lang.ArithmeticException: / by zero
10
10
10
Array Index Error
    java.lang.ArrayIndexOutOfBoundsException:
Index 4 out of bounds for length 4
```

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

  int[] a = {10, 20, 40, 50};
  int[] b = {0, 2, 4, 5, 8};

  for (int i = 0; i <b.length; i++){
    try {
      System.out.println(a[i]/b[i]);
    }
    // catch Arithmetic or Array Index problems
    catch (ArithmeticException | ArrayIndexOutOfBoundsException e) {
      System.out.println("Program Error ");
      System.out.println("\t"+e);
    }
  }
}
```

```
>> java MultiCatch2
Program Error
  java.lang.ArithmeticException: / by zero
10
10
10
Program Error
  java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds for length 4
```

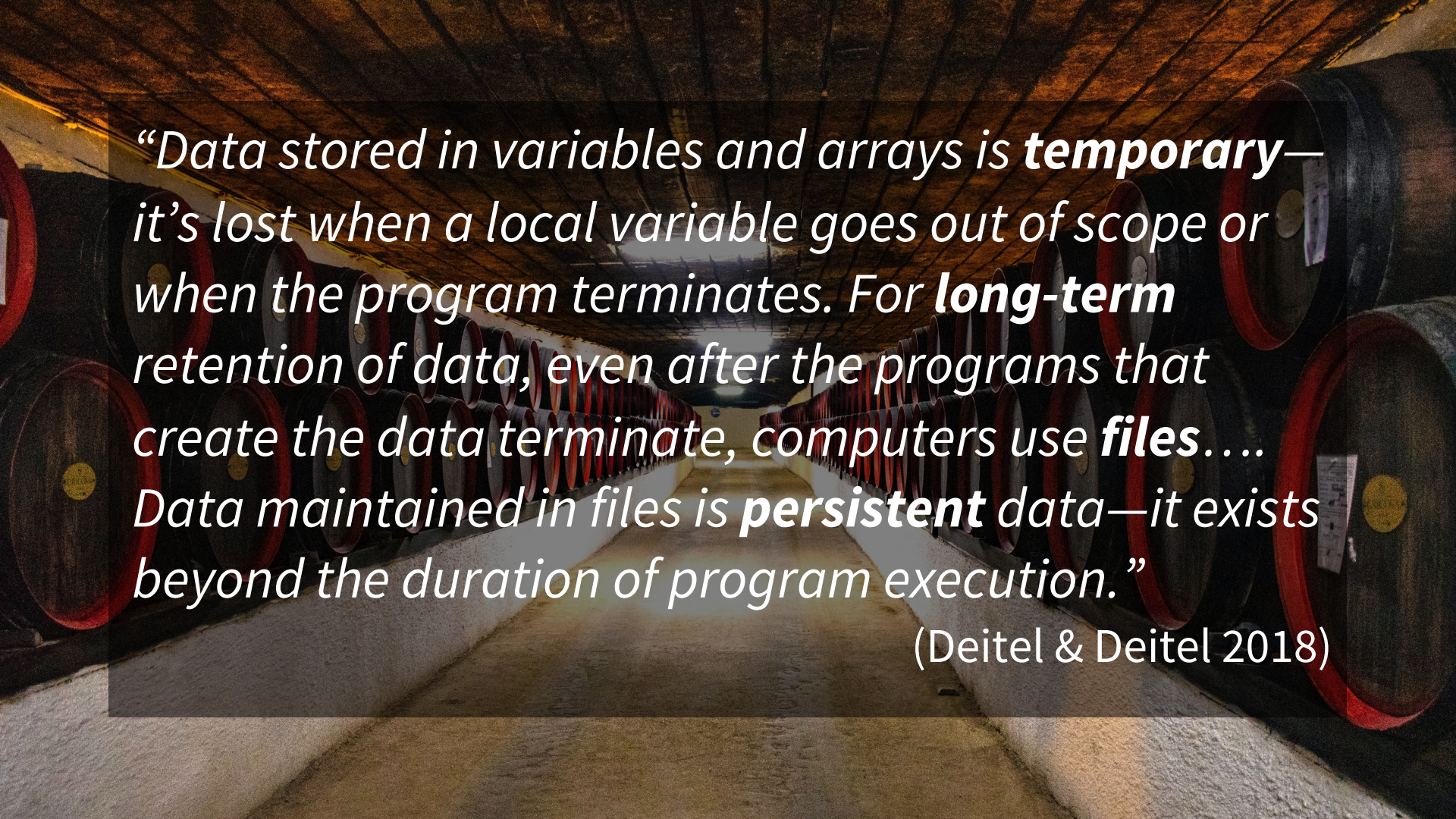
Custom Exceptions

- Exceptions are sub-classes of Exception class
- Exception provides several methods (e.g. `toString()`)
- Custom classes can be created by *extending* Exception
- Permits custom exceptions to be *thrown*
- Exceptions should be the standard way to handle errors

```
// class defines a custom exception
public class Pre2000 extends Exception {
  int year;

  Pre2000(int year) {
    this.year=year;
  }

  @Override
  public String toString() {
    return year + " is less than 2000";
  }
}
```

```java
public class Year2000 {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    try {
      System.out.println("Enter year of 2000 or later: ");
      int year = input.nextInt();
      // throw exception if year < 2000
      if(year<2000) {
        throw new Pre2000(year);
      }
      else{
        System.out.println("A good year.");
      }
    }
    // catch custom Pre2000 exception
    catch (Pre2000 e) {
      System.out.println("Year 2000 Exception");
      System.out.println(e);
    }
    finally {
      input.close();
    }
  }
}
```

```
>> java Year2000
Enter year of 2000 or later:
2023
A good year.

>> java Year2000
Enter year of 2000 or later:
1999
Year 2000 Exception
1999 is less than 2000
```



*"Data stored in variables and arrays is **temporary**—it's lost when a local variable goes out of scope or when the program terminates. For **long-term** retention of data, even after the programs that create the data terminate, computers use **files**.... Data maintained in files is **persistent** data—it exists beyond the duration of program execution."*

(Deitel & Deitel 2018)

File Handling

- Operations adhere to CRUD conventions:
  - Create
  - Read
  - Update
  - Delete

File Handling

- Java has multiple packages and classes to help
- Use of `java.io.File` is a common basis
- Instances of `File` allow handling via its methods
- Also using methods from `FileWriter` and `Scanner` classes
- Take care with directory paths “\” Windows versus “/” MacOS and Linux

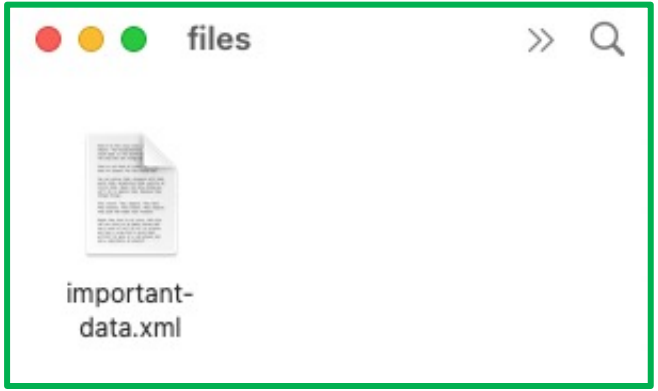
File Handling – Useful Classes & Methods

Operation	Class	Method
Create	File	`createNewFile()`
Read	Scanner	`hasNextLine()` `nextLine()`
Update	FileWriter	`write()`
Delete	File	`delete()`

CreateFile.java

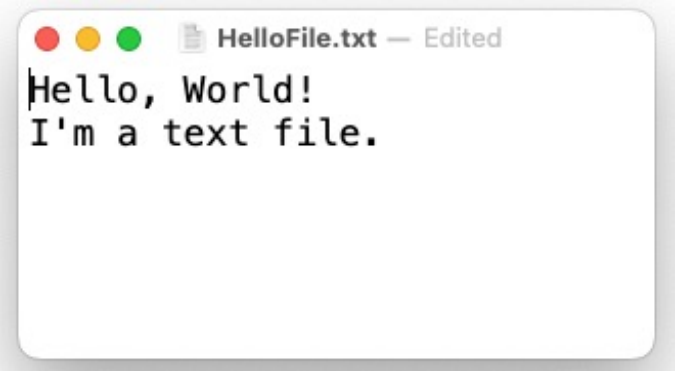
```
static void createFile(String fileName) {
 // note addition of path/folder added
 File dataFile = new File("files/"+fileName);
try {
  // returns true if file created OK
  if (dataFile.createNewFile()) {
   System.out.println("File Created.");
  }
  // else file can't be created
  else {
   System.out.println("Create Error.");
  }
 }
 catch (IOException e) {
  System.out.println("Something went wrong.");
  System.out.println(e);
 }
}
```

```
>> java CreateFile
Enter new file name:
important-data.xml
File Created.

>> java CreateFile
Enter new file name:
important-data.xml
Create Error.
```



ReadFile.java

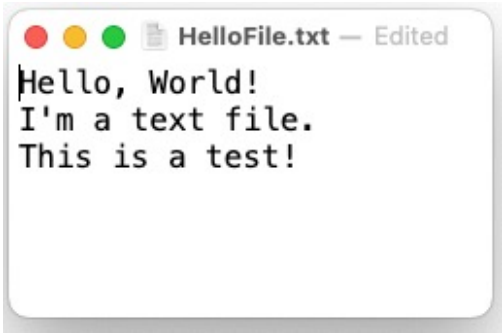
```
static void readFile(String fileName){
 try {
 // note path addition to name
 File dataFile = new File("files/"+fileName);
 // new scanner to read the file
 Scanner read = new Scanner(dataFile);
 // read so long as more data exists
 while (read.hasNextLine()) {
  String data = read.nextLine();
  System.out.println(data);
 }
 read.close();
 }
 catch (FileNotFoundException e) {
  System.out.println("Something went wrong.");
  System.out.println(e);
 }
}
```




```
>> java ReadFile
Enter file to read:
HelloFile.txt
Hello, World!
I'm a text file.
```

UpdateFile.java

```java
static void updateFile(String fileName, String data)
{
 // note path addition to name
 try {
  File dataFile = new File("files/"+fileName);
  FileWriter writer = new FileWriter(dataFile,true);
  writer.write("\n"+data);
  writer.close();
  System.out.println("Data written to file");
 }
 catch (IOException e) {
  System.out.println("Something went wrong.");
  System.out.println(e);
 }
}
```




```
>> java UpdateFile
Enter file to update:
HelloFile.txt
Enter data to add:
This is a test!
Data written to file
```

DeleteFile.java

```
static void deleteFile(String fileName) {
 File dataFile = new File("files/"+fileName);
 if (dataFile.delete()) {
  System.out.println("File Deleted.");
 }
 // else no file access
 else {
  System.out.println("Delete Error.");
 }
}
```

```
>> java DeleteFile
Enter file name to delete:
important-data.xml
File Deleted.
```

Serialization

- Built-in Java *interface*
- Makes file handing large amounts of data easier
- Supports writing and reading *objects*
  - Using `FileOutputStream` and `ObjectOutputStream`
  - Using `FileInputStream and ObjectInputStream`
- Wrapped in exception handling code

SerialFile.java - Writing

```
// write the object to file
try {
 // create instance of the Cat class
 Cat myCat = new Cat("Spot",10,'M');
 FileOutputStream myOutStream = new FileOutputStream("files/dataFile.dat");
 ObjectOutputStream objectOutStream = new ObjectOutputStream(myOutStream);
 objectOutStream.writeObject(myCat);
 objectOutStream.close();
 System.out.println("Serial data written.");
 System.out.println();
}
// note we catch ALL types of exception
catch (Exception e){
 System.out.println("Can't serialize to file.");
 System.out.println(e);
}
```

SerialFile.java - Reading

```
// read in the file and output contents
try {
 FileInputStream myInStream = new FileInputStream("files/dataFile.dat");
 ObjectInputStream objectInStream = new ObjectInputStream(myInStream);
 Cat newCat = (Cat) objectInStream.readObject();
 objectInStream.close();
 // output object contents
 System.out.println("Object Read");
 System.out.println("-----------");
 System.out.println("Name: "+newCat.name);
 System.out.println("Age: "+newCat.age);
 System.out.println("Gender: "+newCat.gender);
}
catch (Exception e){
 System.out.println("Can't read serial from file.");
 System.out.println(e);
}
```

References

Deitel, P. J., & Deitel, H. M. (2018). *Java: how to program : early objects (Global)*. Pearson.

Schildt, H. (2022). *Java: a beginner's guide* (9th ed.). McGraw Hill.

Loy, M. (2020). *Learning Java: an introduction to real-world programming with Java* (Fifth). O'Reilly.