# Exceptions

- Exceptions
  - Introduction
  - Handling exceptions
  - Inbuilt exceptions

# Exceptions - Introduction

- Exceptions are used to handle errors when a method has a requirement but cannot logically continue due to circumstances beyond its control
- Exceptions are used when logically there's no course of action the program can take to continue. E.g.
  - Trying to open a file that doesn't exist
  - Failing to connect to a database
  - Running a database query which is invalid
  - Trying to connect to a database which isn't responding
  - Writing to a file without having write permissions

# Exceptions – The problem they solve

- Consider the following code

```
Scanner s = new Scanner(System.in);
int age;

Person p = new Person();
System.out.println("How old is person 1?");
age = s.nextInt();
p.setAge(age);

Person p2 = new Person();
System.out.println("How old is person 2?");
age = s.nextInt();
p2.setAge(age);
```

# Exceptions - Introduction

- In this example it's possible to set someone's age to an invalid number:
  - 3445540
  - -9
- Exceptions are a form of error handling
- Setting someone's age to one of these values is an error and could result in a program error if a calculation is done on the number

# Exceptions

- To declare that a method can return an exception, you must define it in the method header using the *throws* keyword followed by the exception type (The default is a standard Exception).

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

- The method will now be able to use the *throws* keyword to throw an exception
- All exceptions should happen after conditions (if statements) when an unexpected condition is met:

# Exceptions

- Exception example

```
public static void main(String[] args)  throws Exception {
    int x = 1;
    int y = 2;
    System.out.println("A");
    System.out.println("B");

    if (x < y) throw new Exception("Error");
    System.out.println("C");
}
```

# Exceptions

- Once an exception has been thrown, the method will exit, like a *return* statement.
- Any code after the throw statement will not be run

```
public static void main(String[] args)  throws Exception {
    int x = 1;
    int y = 2;
    System.out.println("A");
    System.out.println("B");

    if (x < y) throw new Exception("Error");
    System.out.println("C");
}
```

- Output:

```
A
B

Exception in thread "main" java.lang.Exception: Error at ExceptionTest.main(ExceptionTest.java:8)
```

- Notice that “C” was never printed and the program halted

# Exceptions

- The person class could be modified to *throw* an exception if an invalid age was entered

# Exceptions

```
public class Person {
    private int age;

    public void setAge(int age) throws Exception {
        if (age < 0 || age > 110) throw new Exception("Invalid Age");
        else this.age = age;
    }
}
```

```
Person p = new Person();
p.setAge(-20);
```

# Exceptions

- This is more useful when collecting input from a user:

```
Scanner s = new Scanner(System.in);

Person p = new Person();

System.out.println("How old are you?");
int age = s.nextInt();
p.setAge(age);
```

- In this case, -20 is a valid integer and will be correctly processed by the scanner.nextInt() line
- It's not a valid age so an exception can be thrown

# Exceptions

- This check could be moved outside the setAge method:

```
Scanner s = new Scanner(System.in);
int age;

Person p = new Person();
System.out.println("How old is person 1?");
age = s.nextInt();
if (age > 0 && age < 110) {
    p.setAge(age);
}


Person p2 = new Person();
System.out.println("How old is person 2?");

age = s.nextInt();
if (age > 0 && age < 110) {
    p2.setAge(age);
}
```

# Exceptions

- This has several disadvantages:

  1) Repeated code - the check has to be made every time a Person's age is set

  2) Person objects can still be created with an invalid age

  3) The check must be manually applied each time. If the class should only allow adults (age >= 18) every instance of the check must be changed

# Exceptions

- The simplest fix is to move the check into the setAge method:

```
public class Person {
    private int age;

    public void setAge(int age) {
        if (age > 0 && age < 110) {
            this.age = age;
        }
    }
}
```

# Exceptions

- This fixes the problem of the person existing with an invalid age but may result in the age never being set

```
Scanner s = new Scanner(System.in);
int age;

Person p = new Person();
System.out.println("How old is person 1?");
age = s.nextInt();
p.setAge(age);
```

- If age is entered as a -9 the program will continue without interruption
- But it may break if a calculation is done on the age later

# Exceptions

- One way of avoiding this is to make the setAge method return true/false depending on whether it's successful

```
public boolean setAge(int age) {
    if (age > 0 && age < 110) {
        this.age = age;
        return true;
    }
    else return false;
}
```

```
Person p = new Person();
System.out.println("How old is person 1?");
int age = s.nextInt();

if (!p.setAge(age)) System.out.println("Invalid age!");
```

# Exceptions

- This solves the problem but it requires the person using the class to understand exactly how it works
- It also allows them to ignore errors and bypass them without problem

```
Person p = new Person();
System.out.println("How old is person 1?");
int age = s.nextInt();

p.setAge(age);
```

# Exceptions

- If there are multiple variables being set, the code becomes very long and repetitive

```
if (!p.setAge(age)) System.out.println("Invalid age!");
if (!p.setName(name)) System.out.println("Invalid name!");
if (!p.setAddress(address)) System.out.println("Invalid address!");
if (!p.setEmail(email)) System.out.println("Invalid email address!");
```

# Exceptions

- Exceptions can be used to overcome these problems
- The setAge method can be rewritten to throw an exception:

```
public class Person {
    private int age;

    public void setAge(int age) throws Exception {
        if (age < 0 || age > 110) throw new Exception("Invalid Age");
        else this.age = age;
    }
}
```

# Exceptions

```
public class Person {
    private int age;

    public void setAge(int age) throws Exception {
        if (age < 0 || age > 110) throw new Exception("Invalid Age");
        else this.age = age;
    }
}
```

- If the age is invalid an exception will be *thrown*
- The code calling the method must be able to *handle* the exception

# Exceptions

- There are two ways of handling the exception
  - 1) Mark the calling method able to throw the exception

```
public static void main(String[] args) throws Exception {
        Scanner s = new Scanner(System.in);

        Person p = new Person();
        System.out.println("How old is person 1?");
        int age = s.nextInt();
        p.setAge(age);

    }
```

# Exceptions

- If the Exception is thrown in the *main* method the program will halt

```
public static void main(String[] args) throws Exception {
        Scanner s = new Scanner(System.in);

        Person p = new Person();
        System.out.println("How old is person 1?");
        int age = s.nextInt();
        p.setAge(age);

    }
```

# Exceptions

- Exceptions can “bubble” up method calls
- If a method is marked to throw the exception, it will get thrown to the method that called it
- If that method is marked to throw the exception it will bubble up to the method that called that

# Person Class

```
public class ExceptionTest {

    public static void main(String[] args) throws Exception {
        method1();
    }


    public static void method1() throws Exception {
        method2();
    }

    public static void method2() throws Exception{
        method3(123);
    }

    public static void method3(int num) throws Exception {
        if (num > 100) throw new Exception("Invalid Number");
    }
}
```

# Person Class

- Once the Exception bubbles out of the main method, the program will halt

```
public class ExceptionTest {

    public static void main(String[] args) throws Exception {
        method1();
        System.out.println("Program continuing to run");
    }


    public static void method1() throws Exception {
        method2();
    }

    public static void method2() throws Exception{
        method3(123);
    }

    public static void method3(int num) throws Exception {
        if (num > 100) throw new Exception("Invalid Number");
    }
}
```

# Exceptions

- If all exceptions did was halt the program they wouldn't be very useful
- You can stop an Exception bubbling at any point by *catching* it
- Exceptions are caught by a block of code known as *try* and *catch*

# Handling Exceptions

- A try block is:
  - One or more statements that are executed
  - and can potentially throw an exception
- The application will not halt if the code in the try block throws an exception
- If code in the try block throws an exception the code in the *Catch* block is executed
- If no exception occurs, all the lines in the try block will be executed but the catch block will never be executed

# Try/Catch

- When using try/catch you do not mark the method as throwing an exception.
- This is because the exception never reaches that far up the call stack
- A try/catch block can catch an exception from any depth

# Try/Catch

```
public class ExceptionTest {

    public static void main(String[] args) {
        try {
            method1();
        }
        catch (Exception e) {
            System.out.println("An exception was caught");
        }
        System.out.println("Program continuing to run");
    }


    public static void method1() throws Exception {
        method2();
    }

    public static void method2() throws Exception{
        method3(123);
    }

    public static void method3(int num) throws Exception {
        if (num > 100) throw new Exception("Invalid Number");
    }
}
```

No “throws” keyword because the exception is caught

The exception will “bubble” up the Call stack

# Handling exceptions

- Going back to the Person example with a valid age range, the original code could be changed to use a try/catch block to handle the error condition

```
public static void main(String[] args)  {
        Scanner s = new Scanner(System.in);

        Person p = new Person();
        System.out.println("How old is person 1?");

        int age = s.nextInt();

        try {
            p.setAge(age);
        }
        catch (Exception e) {
            System.out.println("You entered an invalid age");
        }

        System.out.println("Program continues as normal");
    }
```

# Exceptions

- Because multiple lines of code can be added in a try {} block, you don't need lots of repeated if statements to check each variable has been set correctly

# Exceptions

```
public static void main(String[] args)  {
    Scanner s = new Scanner(System.in);

    Person p = new Person();
    System.out.println("How old is person 1?");



    try {
        int age = s.nextInt();
        p.setAge(age);

        String name = s.nextLine();
        p.setName(name);

        String address = s.nextLine();
        p.setAddress(address);

        String email = s.nextLine();
        p.setEmail(email);
    }
    catch (Exception e) {
        System.out.println("You entered an invalid value");
    }

    System.out.println("Program continues as normal");
}
```

# Exceptions

- The problem with this code is that it's not very helpful for the user. Whatever they get wrong, it shows the message "You have entered an invalid value"
- Exceptions can account for this. When you throw an exception you provide an error message:

```
public class Person {
    private int age;

    public void setAge(int age) throws Exception {
        if (age < 0 || age > 110) throw new Exception("Invalid Age");
        else this.age = age;
    }
}
```

# Catching exceptions

- When an exception is caught you can get the error message from the exception that has been caught
- In the *catch* block of code you provide a variable name that the caught exception is stored in

```
try {
    int age = s.nextInt();
    p.setAge(age);

}
catch (Exception e) {
    System.out.println("You entered an invalid value");
}
```

# Catch block

```
try {
    int age = s.nextInt();
    p.setAge(age);

}
catch (Exception e) {
    System.out.println("You entered an invalid value");
}
```

- Inside the catch block this makes the variable *e* available
- This is a normal variable and can be named anything
- A lowercase *e* is used for exceptions by convention (like i is used in loops)

# Exceptions

- You can read the message that was used in the constructor by calling e.getMessage()

```
try {
        int age = s.nextInt();
        p.setAge(age);

        String name = s.nextLine();
        p.setName(name);

        String address = s.nextLine();
        p.setAddress(address);

        String email = s.nextLine();
        p.setEmail(email);
    }
    catch (Exception e) {
        System.out.println(e.getMessage());
    }
```

# Exceptions

```
try {
        int age = s.nextInt();
        p.setAge(age);

        String name = s.nextLine();
        p.setName(name);

        String address = s.nextLine();
        p.setAddress(address);

        String email = s.nextLine();
        p.setEmail(email);
    }
    catch (Exception e) {
        System.out.println(e.getMessage());
    }
```

- This will now print out the text of the exception that was caught

# Exception types

- There are many different types of exception. Some examples are:
  - The generic *Exception* class I've used so far
  - *NumberFormatException* which is thrown when you try to format a string as a number
  - *ArrayIndexOutOfBoundsException* which is thrown when you try to write to an array index that doesn't exist
  - *FileNotFoundException* which is thrown when a file needed cannot be found

# Catch block

- You can use a catch block to catch specific exception types.

```
try {
    Integer.parseInt("ABC");
}
catch (NumberFormatException e) {
    System.out.println(e.getMessage());
}
```

# Catch block

- This will **only** catch exceptions of the specified type.

```
int[] intArray = new int[3];

try {
    intArray[999] = 123;
}
catch (NumberFormatException e) {
    System.out.println(e.getMessage());
}
```

This will generate a ArrayIndexOutOfBoundsException

But it won't be caught by this catch block because it can only catch NumberFormatExceptions

# Catch block

- Catching *Exception* will allow you to catch any type of Exception

```
int[] intArray = new int[3];

try {
    intArray[999] = 123;
}
catch (Exception e) {
    System.out.println(e.getMessage());
}
```

This will generate a ArrayIndexOutOfBoundsException

Will be caught because the Catch block is set to catch Any Exception type

# Catch block

- You can provide multiple catch blocks

```
try {
    intArray[999] = 123;
}
catch (NumberFormatException e) {
    System.out.println(e.getMessage());
}
catch (Exception e) {
    System.out.println(e.getMessage());
}
```

This code will run if a NumberFormatException is caught

This code will run if any other Exception type is caught

# Catch block

- The order of the catch blocks matter and run in order

```
try {
    intArray[999] = 123;
}
catch (Exception e) {
    System.out.println(e.getMessage());
}
catch (NumberFormatException e) {
    System.out.println(e.getMessage());
}
```

This code will run if any exception Is caught

This code will never run Because all exceptions will be Caught by the catch block above

# Catch block

- A try statement may only have one catch clause for each type of exception

```
try {
    intArray[999] = 123;
}
catch (NumberFormatException e) {
    System.out.println(e.getMessage());
}
catch (NumberFormatException e) {
    System.out.println(e.getMessage());
}
```

Invalid! A Catch block for NumberFormatException Already exists!

# Exceptions – Why are they used?

- Exceptions are used because unexpected things do happen (Especially when asking for input from the user!)
- They allow you as the programmer to decide how to recover from an error that would otherwise be unrecoverable or cause problems in the program

# Exceptions

- For example, if you wanted to write something to a file and it didn't work (invalid file name, no permissions, file locked, etc)
- If nothing happened and it looked like it saved successfully the program would break once the file was re-opened

# Exceptions

- Exceptions allow the program to provide error-recovery in the catch block.
- If the user tries to save a file and it doesn't work you could ask them to choose a different location

# Exceptions in loops

- Exceptions can be used inside loops for this purpose. Consider the person example from earlier. You can put this inside a loop to keep asking the user for an age until they enter a valid one:

```
Person p = new Person();
boolean validAge = false;

while (!validAge) {
    try {
        System.out.println("How old is person 1?");
        int age = s.nextInt();
        p.setAge(age);
        validAge = true;
    }
    catch (Exception e) {
        System.out.println(e.getMessage() + " Please try again" );
    }
}
```