# ENTERPRISE JAVA PROGRAMMING / PROGRAMMING II

MODULES CT545 / CT875, SEMESTER 2, ACADEMIC YEAR 2020-2021
NATIONAL UNIVERSITY OF IRELAND, GALWAY

*Lecture 1b*

Lecturer: Dr Patrick Mannion

# TOPICS

- More about Polymorphism and reference types
- Abstract classes, interfaces, Java 8 default methods
- Exception handling, vs. Java 8's `Optional` class

# MORE ABOUT POLYMORPHISM

- As you know, all variables (including method parameters) and values of expressions (including method results) have a *type*
- There are *primitive types* and *reference types* in Java
- Primitive types are, e.g., `int`, `float`, `char`, ...
- Reference types are classes or interfaces
- E.g., in `String s = "bleep"+"blob";`
  the type of variable `s` and the type of `"bleep"+"blob"` are `String`.

# MORE ABOUT POLYMORPHISM

- If the reference type is a class, it is also called *class type*
- If the reference type is an interface, it is sometimes called *interface type*
- If the reference type is an array, it is also called *array type*
- But you can safely use the term "reference type" in all those cases

# MORE ABOUT POLYMORPHISM

- Primitive types such as `int`, `float`, `double` or `char` don't have methods (because they are not classes).
- Since Java 5, there are so-called *primitive wrapper classes* (or "wrapper classes") which can be used in place of primitive types.

| Primitive type | Wrapper class |
| --- | --- |
| `byte` | `Byte` |
| `short` | `Short` |
| `int` | `Integer` |
| `long` | `Long` |
| `float` | `Float` |
| `double` | `Double` |
| `char` | `Character` |
| `boolean` | `Boolean` |

- Conversion between primitive types and their respective wrapper classes is done automatically (so-called *boxing* and *unboxing*):

```
Integer someInteger = 5;   // boxing

int someInt = Integer(7);   // unboxing
```

# MORE ABOUT POLYMORPHISM

- The type of a variable indicates the kind of "thing" that can be stored in this variable
- If a variable should be used for objects (vs. primitive values), it needs to have a reference type

# MORE ABOUT POLYMORPHISM

- The purpose of types is mainly to ensure (part of) the *correctness* of a program
- If we assign the value of an expression to a variable, the type of the variable and the type of the expression need to be *type-compatible*
- Type-compatibility is checked at compile time (that is, <u>before</u> runtime). Therefore, Java is a *statically typed* language. This check is called *type checking*.
- E.g.,
  - `int x = "Hello!";` → <u>not</u> type-compatible, error
  - `int x = 123*456;` → type-compatible, fine

# MORE ABOUT POLYMORPHISM

- Let’s look at the class `Circle` from the previous lecture again, focusing on methods which override methods of the superclass:

# MORE ABOUT POLYMORPHISM

```
class Circle extends GeometricFigure {

   private double radius;

   public Circle(double radius) {
      this.radius = radius;
   }

   public double calcArea() {
      return radius*radius*PI;
   }

   public void display() {  // overrides method display() in superclass
      System.out.println("This is a circle with radius " + radius);
   }
}
```

# MORE ABOUT POLYMORPHISM

```
class GeometricFigure {
   public static final double PI = 3.141592653;
   private boolean filled;

   public GeometricFigure() {
      filled = false;
   }
   public boolean isFilled() {
      return filled;
   }
   public void setFilled(boolean filled) {
      this.filled = filled;
   }
   public void display() {  // overridden in class Circle
      System.out.println("This is some geometric figure.");
   }
}
```

# MORE ABOUT POLYMORPHISM

- The following assignment works fine:

```
GeometricFigure g = new Circle(12.3);
```

- Generally, we can assign objects to variables in case the type of the variable is the same class or a superclass of the class of the expression at the right-hand side of the assignment operator `=`
- In the example, the type of the right-hand side (`Circle`) is *type-compatible* with the type of variable `g` (`GeometricFigure`)

# MORE ABOUT POLYMORPHISM

- The other way round (`Circle c = new GeometricFigure()`) would not be allowed, and also not something like `Circle c = new Rectangle(10.1,1.9)`
- We could try to *cast* the right-hand side to type `Circle`:

```
Circle c = (Circle) new GeometricFigure();
```

- But this would lead to a runtime-error, because the object created with `new GeometricFigure()` isn't actually a circle.
- General rule: avoid type casts as far as possible!

# MORE ABOUT POLYMORPHISM

- As you know, it's also possible to pass arguments which have reference type `X` to methods where the respective formal parameter has a superclass of `X` as its reference type. E.g.,

  Method declaration:

```
void someMethod(int p1, GeometricFigure g) {
   …
 }
```

  Method call:

```
someMethod(55, new Circle(12.3)); /* works fine
  because Circle is a subclass of GeometricFigure */
```

# MORE ABOUT POLYMORPHISM

- We say that a variable is *polymorphic* if it can store references of objects with several different types.
- E.g., `GeometricFigure g = new Circle(12.3);`

- Variable `g` is polymorphic because it can store objects of reference types `Circle`, `Rectangle` or `GeometricFigure`
  (that is, any object whose class is `GeometricFigure` or a subclass of `GeometricFigure`)

# MORE ABOUT POLYMORPHISM

- After

  `GeometricFigure g = new Circle(12.3);`

  we have a variable `g` which has reference type `GeometricFigure` but actually refers to an object of class `Circle`...
- So it seems that, *in some sense*, the type of `g` (which is `GeometricFigure`) and the class of the object stored in `g` (which is `Circle`) "disagree".
- Which method would

  `g.display()` call? The method declared in class `GeometricFigure` or the method declared in class `Circle`?
  We already now the answer, but how does Java handle this case in detail...?

# MORE ABOUT POLYMORPHISM

- `g.display()` calls the method declared in class `Circle`. As expected, Java calls the method declared by the most *specific* class of the object in `g`, which is `Circle`.
- We say that the *static type* of `g` is reference type `GeometricFigure`, whereas the *dynamic type* of `g` is reference type `Circle`.
- Java determines the dynamic type at runtime and uses this type to call the right method, i.e., to resolve overriding. This approach is called *dynamic dispatch* (a.k.a. *dynamic binding*).

# MORE ABOUT POLYMORPHISM

- The behavior of `g = new Circle(…)` and `g.display()` is not complicated or surprising at all.
- But could we also use variable `g` in order to call a method which is declared *only* in `Circle`? No.

```
Circle myCircle = new Circle(11.8);
GeometricFigure g = new Circle(9.2);

myCircle.calcArea();  // works
g.calcArea(); // illegal, even though g references a circle
```

- I.e., dynamic dispatch works only for overriding methods!
- Java requires that *at compile time*, the right side of the dot-operator (i.e., `calcArea()`) is a member of the class which is the <u>statically declared type</u> of the object. But here, the static type of `g` is `GeometricFigure` (which doesn't have a `calcArea()`).

# MORE ABOUT POLYMORPHISM

- You can use the following rule to determine which method Java will call:
  "*If an object o is an instance of classes $C_1$, $C_2$, $C_3$ ,…, $C_n$, and $C_2$ is a superclass of $C_1$, $C_3$ is a superclass of $C_2$ and so on, and we call a method m of this object, Java searches at runtime for m within $C_1$ first, then in $C_2$ , then in $C_3$ and so on, and then calls the version of m which was found first.*"
- However, in most practical cases, it is more intuitive to use the "overriding" metaphor (Java calls the method which has "overridden" all methods in the superclasses)

# ABSTRACT CLASSES AND INTERFACES

- Looking once again at the classes `Circle` and `Rectangle`, we focus again on the methods which calculate the area:

# ABSTRACT CLASSES AND INTERFACES

```
class Circle extends GeometricFigure {

   private double radius;

   public Circle(double radius) {
      this.radius = radius;
   }

   public double calcArea() {
      return radius*radius*PI;
   }

   public void display() {
      System.out.println("This is a circle with radius " + radius);
   }
}
```

# ABSTRACT CLASSES AND INTERFACES

```
class Rectangle extends GeometricFigure {

   private double width;
   private double height;

   public Rectangle(double width, double height) {
      this.width = width;
      this.height = height;
   }

   public double calcArea() {
      return width*height;
   }
}
```

# ABSTRACT CLASSES AND INTERFACES

- Actually, all kinds of *concrete* geometric shapes, such as circles and rectangles, have an area that can be calculated. Both methods even have the same signature.
- So, if we have a variable for geometric figures, it seems to be reasonable to determine the area for this variable.
- But something like the following would <u>not</u> work, as we have seen on previous slides already (because there is no `calcArea` in `GeometricFigure`):

```
GeometricFigure g = null; // at this point we don't know yet which
// concrete geometric shape (e.g., might depend on user input)
...
g = new Circle(123.4);  // now we know which shape (a circle)
double a = g.calcArea();  // wrong
```

# ABSTRACT CLASSES AND INTERFACES

- One way to overcome this problem would be to make an explicit type cast:

```
GeometricFigure g = null;
...
g = new Circle(123.4);
double a = ((Circle)g).calcArea();  // works
```

- This cast is a so-called *downcast* (where the resulting type is more specific than the original type). The static type of `(Circle)g` is `Circle`, which has a method `calcArea`.
- But the problem with this solution is that we need to know at compile time which type to cast to. If, e.g., the class depends on user input, and the user chooses at runtime to store a rectangle in `g`, the above type cast would not work correctly (error at runtime).

# ABSTRACT CLASSES AND INTERFACES

- Therefore, a much better solution is to add a method `calcArea` to class `GeometricFigure`.
- This would have (at least) two benefits:
  1) It would nicely reflect the reality that all geometric figures have an area.
  2) It would allow us to create variables with reference type `GeometricFigure` and then call the method `calcArea` on these variables – even *without knowing* what concrete kind of figure (a circle or a rectangle or …) is actually stored in these variables (thanks to polymorphism and dynamic binding!).

# ABSTRACT CLASSES AND INTERFACES

- But we are facing a problem when we actually want to declare this method:

```
class GeometricFigure2 {
    …
    public double calcArea() {
      return ???;  // What shall this method do?
    }
    …
```

- There is no way to compute the area unless we know what concrete kind of figure we are talking about.

# ABSTRACT CLASSES AND INTERFACES

- Every object of class `GeometricFigure2` has an area, but unless we also know that it is a circle or a rectangle, there is no way to determine it concretely.
- Again, you likely know the solution for this issue already…

# ABSTRACT CLASSES AND INTERFACES

- We need a way to state that a method `calcArea` is declared in class `GeometricFigure2` without being forced to specify a body for this method within `GeometricFigure2`.
- As you already know, Java supports this in form of *abstract methods*, which are declared as follows:

```
public abstract double calcArea();  // no body!
```

- Abstract methods are essentially signatures without a body.
("body" = the code inside the {curly brackets} of a method)

# ABSTRACT CLASSES AND INTERFACES

- If a class declares at least one abstract method, the whole class becomes a so-called *abstract class*.
- Abstract classes need to be declared using the `abstract`-modifier in the header of the class declaration:

```
abstract class GeometricFigure2 {
          …
    public abstract double calcArea();
    …
}
```

# ABSTRACT CLASSES AND INTERFACES

```
abstract class GeometricFigure2 {
    public static final double PI = 3.141592653;
    private boolean filled;

    public GeometricFigure2() {
       filled = false;
    }
    public boolean isFilled() {
       return filled;
    }
    public void setFilled(boolean filled) {
       this.filled = filled;
    }
    public void display() {
       System.out.println("This is some geometric figure.");
    }
    public abstract double calcArea();
}
```

# ABSTRACT CLASSES AND INTERFACES

- Abstract classes can declare non-abstract methods too. You can mix abstract and normal methods freely.
- A class can even be made abstract (non-instantiable) without declaring any abstract methods at all, just by using the modifier `abstract` before `class`.

# ABSTRACT CLASSES AND INTERFACES

- Abstract classes only really make sense if concrete classes (i.e., of which you can create objects) are derived from them.
- So let's assume class `GeometricFigure2` is the new superclass of `Circle` and `Rectangle`
- Methods `calcArea` in classes `Circle/Rectangle now` override the abstract method `calcArea` in `GeometricFigure2`

# ABSTRACT CLASSES AND INTERFACES

- It is not possible to create a direct instance of an abstract class:

```
GeometricFigure2 g = new GeometricFigure2(); // illegal
```

- We can only create objects from a concrete (that is, non-abstract) subclass of an abstract class:

```
GeometricFigure2 g = new Circle(4.2); // fine
```

(assuming that `Circle` is derived from `GeometricFigure2`)

# ABSTRACT CLASSES AND INTERFACES

- A subclass of an abstract class must either implement all abstract methods of the superclass, or it must be declared abstract itself.
- It is thus possible that a subclass of an abstract class is abstract itself.
- It is also possible that a superclass is concrete (non-abstract) and one of its subclasses is abstract.

# ABSTRACT CLASSES AND INTERFACES

- After adding the abstract method `calcArea` to the geometric figures class, we can do the following:

```
GeometricFigure2 g = null;

int userChoice = userInput();

if(userChoice==1)
    g = new Circle(12.3);
else
    g = new Rectangle(9.2, 7);
double area = g.calcArea();
```

# ABSTRACT CLASSES AND INTERFACES

- The call of `calcArea` in the last statement

  ```
  double area = g.calcArea();
  ```

  works fine because `g` is a polymorphic variable. Java looks at runtime which most specific class the object stored in `g` actually has (`Circle` or `Rectangle`, depending on user input). Then it calls the variant of `calcArea` which is declared in the discovered class.

# ABSTRACT CLASSES AND INTERFACES

- A class-like entity which can be seen as a sort of "ultimate abstract class" is the *interface*.
- Interfaces are declared as follows:

```
interface MyInterface {
    ...members
}
```

- Until Java 8, an interface had <u>only</u> abstract methods and no fields other than static final fields (i.e., constants).
- Furthermore, the only allowed visibility modifier for members is `public`.
- No constructors allowed in interfaces.

# ABSTRACT CLASSES AND INTERFACES

- Since Java 8, interfaces also allow for static methods and *default methods.*
- Example (inside the body of some interface):

```
default void show(String message) {
    System.out.println(message);
}
```

- If a class implements this interface with default method `show(String message)`, this method is called when `show("…")` is called on an object of the class.
- Default methods can be overridden in the implementing classes:
  If a class implements the interface and doesn't declare a method with the same signature as the default method, the default method is called if a method with that signature is called on an object of the class.
  Otherwise, the method defined in the class overrides the default method in the interface, i.e., the method in the class is called (see above).

# ABSTRACT CLASSES AND INTERFACES

- As you know, a certain class can implement more than one interface.
- Such "poor-mans multiple inheritance" is still allowed even with default methods, however, if a class implements two or more interfaces which define default methods with identical signatures (e.g., both interfaces have a default method show(String …)), the class must itself provide an implementation of a method with that signature (that is, the default methods in the interfaces won't be used).

# ABSTRACT CLASSES AND INTERFACES

- To "derive" a class from one or more interfaces, the keyword `implements` is used (not `extends`):

  ```
  class MyClass implements Interface1, Interface2 {
       …
  }
  ```

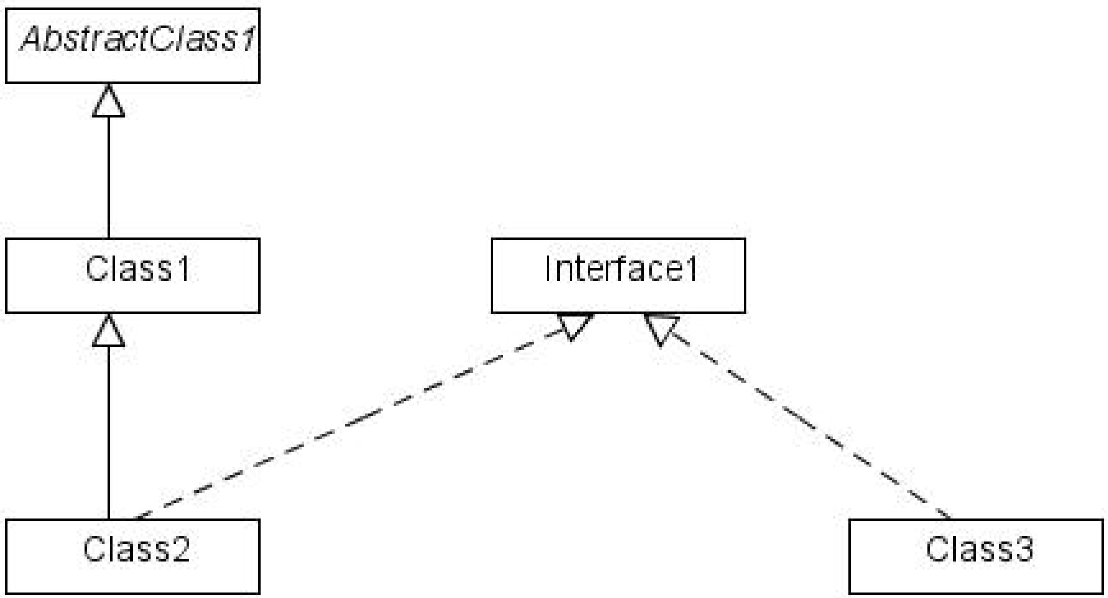
  Also allowed:

  ```
  class MyClass extends MySuperclass
       implements Interface1, Interface2 {
       …
  }
  ```

- A class which implements an interface needs to implement all methods declared by the interface, or this class needs to be an abstract class.

# ABSTRACT CLASSES AND INTERFACES

- The relation between a class and an interface is known as *weak is-a* relationship.




- Abstract classes are distinguished from normal classes by the italic-printed class name. Interfaces are sometimes drawn as circles, but this is not a fixed convention.

# ABSTRACT CLASSES AND INTERFACES

- Apart from the explained differences, interfaces can be used more or less like classes. E.g., one can
  - use them as the reference type of a variable. E.g.

    `MyInterface mI = new MyClass();`
    (provided `MyClass` implements `MyInterface`);
  - use `object instanceof MyInterface` to test whether an object belongs to a class which implements this interface.
  - derive an interface from another interface using `extends`(!). It is even possible to let an interface be "sub-interface" of multiple interfaces:

```
interface Interface1 extends Interface2, Interface2 {
   …
}
```

# ABSTRACT CLASSES AND INTERFACES

- Interfaces are widely used, and have many purposes:
  - To achieve a restricted form of multi-inheritance…
  - To separate the publicly accessible aspects of a class from its implementation and non-accessible members. The interface is all a user needs to know in order to use the implementing class.
  - To demand a certain functionality for several different classes without fixing its implementation.
  - To be used instead of an abstract class where all methods are abstract, in order to emphasize the abstractness.
  - As a *marker*. That is, a class implements a possible empty interface in order to be marked as having a certain property. We will later see examples for this use case.

# ABSTRACT CLASSES AND INTERFACES

- Example: The `Comparable` interface (-->Semester 1)
- Many algorithms and build-in Java methods require the comparison of two objects (e.g., sorting algorithms).
- The comparison shall find the "order" of these objects. E.g., "abc" 〈 "xyz"
- However, there is no sensible single way to achieve this (e.g., two strings are compared differently than two circles or two numbers).

# ABSTRACT CLASSES AND INTERFACES

- This is a typical application for an interface. Java defines the build-in interface `Comparable` which is implemented by all classes which provide a comparison method with a defined signature and semantics.

```
public interface Comparable {
   public int compareTo(Object o);
}
```

- The implementations of method `compareTo` within the classes which implement this interface shall result in a negative integer if object `this` is "less" than object `o`, in zero if both objects are equal, and a positive integer otherwise. In other words, it imposes an *order* on the compared objects.

# ABSTRACT CLASSES AND INTERFACES

- E.g., to make two rectangles comparable, we can declare the `Rectangle` class as implementing interface `Comparable` and adding the following method.

```
class Rectangle extends GeometricFigure2 implements Comparable {
     public int compareTo(Object o) {
          if(calcArea() > ((Rectangle)o).calcArea())
               return 1;
          else if(calcArea() < ((Rectangle)o).calcArea())
               return -1;
          else
               return 0;
     } ...
```

Could this code be simplified?

Yes, by using the generic version of the interface - Comparable<T>

We'll discuss generics in more detail later!

# ABSTRACT CLASSES AND INTERFACES

- Implementing this interface allows to pass objects of class `Rectangle` to several build-in Java methods which expect parameters of reference type `Comparable`, such as several *sorting* methods defined in the Java API (or to your own "`Comparable` –aware" methods). Also: *binary search* in sorted sequences.
- Many build-in classes such as `String` or `Date` (a class for calendar dates) already implement `Comparable`

# EXCEPTION HANDLING

- Another Java 8 addition: `Optional`

- To motivate this, let's look at *exception handling* in Java...

# EXCEPTION HANDLING

## Why exceptions?

- Exception handling provides an elegant mechanism for dealing with abnormal conditions at runtime.
- Most important: handling of errors and unexpected situations, such as invalid arguments.
- An exception signals...
  - that something went wrong.
  - that the normal "flow" of the program *cannot* be continued.

# EXCEPTION HANDLING
## Why exceptions?

- Imagine the following situation:
  - For your method *m*, a certain argument e should not be allowed.
  - But for some reason outside of your control, you sometimes cannot prevent that *m* is invoked with e as argument.
- You need to take care that the caller of *m* is notified about this problem <u>and</u> that method m doesn't continue using wrong input (i.e., value e).

# EXCEPTION HANDLING

## Why exceptions?

- You could do this by letting your method return a special value which indicates the abnormal condition (e.g., `null`).
- Possible problems with this approach
  - Either it doesn't tell anything about the error (e.g., in case `null` is used)
  - Or you would need to return a complex object which contains both the normal result and the error condition.
  - The caller would need to check for this special error value, which cannot be enforced.
  - If you return a primitive typed value (e.g., `int`), you would need to "misuse" a normal value such as -1 or 9999 (so-called "magic number", considered to be messy code)

# EXCEPTION HANDLING

## Example

Without exceptions, error indication could look like this:

```
public ContactDetails getDetails(String key) {

   if(key == null)
      return null;  // bad practice

    return contacts.get(key);
}
```

# EXCEPTION HANDLING

## Why exceptions?

- Compared to the naive approach (checking for null) a better solution for Java <= 7 is most of the time *throwing an exception*
- No special return value to indicate an error required.
- Errors cannot simply be ignored by the caller.
- The normal program flow is interrupted immediately.

- Many build-in Java methods might throw exceptions which you need then to *catch* and handle (react upon their occurrence).

- Nevertheless, exceptions are nowadays seen as a controversial feature, because they disrupt the normal control flow.

# EXCEPTION HANDLING
## Example

Throwing an exception:

```
public ContactDetails getDetails(String key) {

    if(key == null) {
        throw new NullPointerException(
                        "null key in getDetails");
    }
    return contacts.get(key);
}
```

# EXCEPTION HANDLING

## Example

- As you can see, exceptions are objects of certain *exception classes*.
- They are all subclasses of class `Exception` or class `Error`, which are derived from class `Throwable`.
  `Error` is in fact an exception class too, but reserved for system exceptions thrown by the Java interpreter directly.
- Java pre-defines several of these classes, but you can also define your own exception classes by creating appropriate subclasses.

# EXCEPTION HANDLING

- What happens after an exception has been thrown by a method?
  - The throwing method finishes prematurely.
  - No return value is returned.
  - Control does not return to the caller's point of call!
  - The caller may perform *exception handling* code (so-called *catching* of the exception).
  - If it is a so-called *checked exception*, the exception <u>must</u> be caught by the caller, or "propagated"…
  - In contrast, *unchecked exceptions* cause program termination if not caught.

# EXCEPTION HANDLING

## Checked exceptions

- If a method might throw a checked exception, this needs to be declared:

```
public void saveToFile(String filename)
        throws FileHandlingException
```

- The method caller must catch this exception then, or throw it himself (the so-called "propagation" of the exception).
- Use it for anticipated errors, where recovery may be possible.
- E.g., `IOException` is a build-in class for checked exceptions.
- Throw checked exceptions with care (programmers tend to overuse checked exceptions in Java)
- Java 8: As an alternative to exception throwing, consider `Optional<T>` … (explained in a few minutes)

# EXCEPTION HANDLING

## Unchecked exceptions

- Unchecked exceptions do not need to be caught (but can be).
- Used for unanticipated errors (typically very severe errors)
- Where recovery is unlikely (terminate the program instead).
- E.g., `NullPointerException`

# EXCEPTION HANDLING

Catching an exception (so-called *exception handling*):

```
try {

    In here an exception might be thrown, so we need to enclose
    this code piece with try { … }

} catch(SomeKindOfException ex) {
    Report error. If possible, recover from the error.

} finally {  // the finally branch is optional
    Perform any actions here common to whether or not
    an exception has been thrown.
    Will always be performed, even if there is, e.g., a
    return statement in the try or catch block (!)
}
```

# EXCEPTION HANDLING
## Example

- Example:

1. Exception thrown in saveToFile

```
try {

    addressbook.saveToFile(filename);
    tryAgain = false;

} catch(FileHandlingException e) {

    System.out.println("Unable to save to " + filename);
    System.out.println("Exception: " + e); // info about e
    e.printStackTrace(); // even more info

    tryAgain = true;

}
```

2. If an exception has been thrown, control transfers to here

# OPTIONAL

## Example

Since Java 8, there is a better approach available for many use cases: `Optional`

```
import java.util.Optional;

public Optional<ContactDetails>
    obtainDetails(Optional<String> keyOpt) {
    if(keyOpt.isPresent())
            return Optional.of(contacts.get(keyOpt.get()));
    else
            return Optional.empty();
}
...
Optional<ContactDetails> cdOpt =
    obtainDetails(Optional.of("Tom"));
```

Note: While an improvement over *null* and exception handling, this is still clumsy compared to similar concepts in functional programming languages like Haskell

# OPTIONAL

- Technically a Java 8 Optional value is a "container" (a kind of set) which is either empty or contains a value (a single other object).
- If the `Optional` (the "container") is empty, this indicates that the value is missing (i.e., similar purpose as `null`), e.g.,
  in case of an error which prevented computing the missing value.
- No `null` involved here at all, thus no null-pointer exceptions can occur anymore. If an empty `Optional` value is used in a computation, the computation simply skips it.
- Also, we don't need (most) exceptions anymore - instead we wrap results of computations which might go wrong in Optionals, with an empty `Optional` indicated an error (e.g., failed file access).
- Thus, `Optional` addresses thus two issues:
  - errors can often be handled without the need to throw or handle exceptions
  - replaces most of the time the need to use `null` in a program

# OPTIONAL

- Optionals are used with type arguments. An object of type `Optional<T>` encapsulates a value of type T or is empty
- `isPresent()` checks if the optional object is empty or not
- `get()` obtains the value "inside" the optional object. If no value present, an exception is thrown.
  A better approach is the following (better than `get`):
- `orElse(defaultValue)` (why is this preferred over `get`?)
- Several other "Optional-aware" methods exist in Java >=8, e.g., `filter`. Most of them have in common that they can handle <u>both</u> non-empty and empty Optionals, thus they automatically deal with errors indicated by empty Optionals (or values which are missing for other reasons). No need for `null` or other "magic values" such as -1 here.
- See API documentation for `Optional` for details

# OPTIONAL

A simpler example:

```
Optional<String> nameOpt = Optional.of("John");
```

Creates a non-empty Optional which contains value "John"

```
System.out.println("Details: " + nameOpt.orElse("(unknown name)"));
```

Gives the object inside the Optional `nameOpt`, or, in case the Optional is empty, the fallback value "(unknown name)"

# OPTIONAL

```
...
Optional<String> nameOpt = Optional.empty();
```

Creates an empty Optional because the person's name is unknown for some reason (e.g., in case there was some error before when retrieving the name, or as an initial or default value)