Exceptions

Recap

- Exception is a class.
- Java comes with many, we can write our own.
- The Exception objects, along with some Java-specific structures, allow us to manage exceptional cases in our code.
- These keywords make the mechanism:
  - `try, catch, throw, throws`
- There are two kinds of exceptions:
  - Checked (non-runtime): occur when we can anticipate the error.
  - Unchecked (runtime): when we can't anticipate (bug).
  - (a third type called "error" is not part of the syllabus).

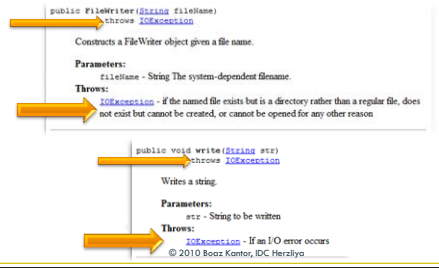
Catching

- Consider the following method:

```
public static void saveTextToFile(String filename, String text) {
        FileWriter writer = new FileWriter(filename);
        writer.write(text);
}
```

- What happens if:
  - There is already a read-only file with that name?
  - There is a folder with that name?
  - The program encounters security difficulties writing the text in the file?
  - There is any other problem of opening such a file for writing?
  - The hard disk is full?
  - **What happens if the methods can't do what they are intended to do?**

Let's look at the API



```
public FileWriter(String fileName)
        throws IOException
```

Constructs a FileWriter object given a file name.

Parameters:
fileName - String The system-dependent filename.

Throws:
IOException - if the named file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason

```
public void write(String str)
        throws IOException
```

Writes a string.

Parameters:
str - String to be written

Throws:
IOException - If an I/O error occurs

Catching

- We should catch these exceptions, like this:

```
public static void saveTextToFile(String filename, String text) {
        FileWriter writer = null;
        try {
                writer = new FileWriter(filename);
        } catch (IOException e) {
                System.out.println("Can't open file for writing");
                return;
        }
        try {
                writer.write(text);
        } catch (IOException e) {
                System.out.println("Can't write to file");
                return;
        }
}
```


Catching

- We can include several instructions in the same try block:

```
public static void saveTextToFile(String filename, String text) {
        FileWriter writer = null;
        try {
                writer = new FileWriter(filename);
                writer.write(text);
        } catch (IOException e) {
                System.out.println("Can't open file or write to file");
                return;
        }
}
```

- 'e' holds a reference to the exception object, which was instantiated (and thrown) by the throwing method.

More examples

- Write a method ***safeCharAt(String s, int i)*** that returns the character located at index ***i*** in ***s***.
- Sounds easy:

```
public static char safeCharAt(String s, int i) {
      return s.charAt(i);
}
```


- But let's look at the API of `String.charAt()`:

Note there is no "throws" section in the method signature!

`public char charAt(int index)`

Returns the `char` value at the specified index. An index ranges from `0` to `length() - 1`. The first `char` value of the sequence is at index `0`, the next at index `1`, and so on, as for array indexing.

If the `char` value specified by the index is a surrogate, the surrogate value is returned.

Specified by:
`charAt` in interface `CharSequence`

Parameters:
`index` - the index of the `char` value.

Returns:
the `char` value at the specified index of this string. The first `char` value is at index `0`.

Throws:
`IndexOutOfBoundsException` - if the `index` argument is negative or not less than the length of this string.

But it seems that the method may throw an exception after all.
Let's go to the API of IndexOutOfBoundsException

Runtime (unchecked) exceptions

- Here is what we learn from this:
  - java.lang: No need to import anything.
  - Hierarchy: top is general, bottom is specific. So:
    - Everything is an Object
    - We can throw this object
    - It's a throwable object of type exception
    - **It's a runtime exception**
  - There are more runtime exceptions of type `IndexOutOfBoundsException`.

java.lang

Class IndexOutOfBoundsException

java.lang.Object
└ java.lang.Throwable
└ java.lang.Exception
└ java.lang.RuntimeException
└ java.lang.IndexOutOfBoundsException

All Implemented Interfaces:
Serializable

Direct Known Subclasses:
ArrayIndexOutOfBoundsException, StringIndexOutOfBoundsException

Catching a runtime exception

- Java allows us not to catch it. Consider the following `main` method:

```
public static void main(String[] args) {
    System.out.println(safeCharAt("Hello, world!", -1));
}
```

```
public static char safeCharAt(String s, int i) {
    return s.charAt(i);
}
```

- What happens here, is that `s.charAt(i)` throws an `IndexOutOfBoundsException`, but since `safeCharAt` doesn't catch it, it is rethrown to the calling method, `main`.

Ignoring exceptions

- We can always ignore an exception and have it rethrown to our calling method.
- We then become the throwing method.
- By doing so, we take the risk of exceptions reaching the main method.
- If the main method throws an exception, our program crashes.
- We wish to avoid that.
- As a rule, **we always catch an exception in a method that is, logically, the one in charge of handling such an exception.**

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

```
public void f() {
    g();
}
```

```
public void g() {
    safeCharAt("Hello, world!", -1);
}
```

```
public static char safeCharAt(String s, int i) {
    return s.charAt(i);
}
```

```
public char charAt(int i) {
    // throws an exception
}
```


Where to catch them?

- As a rule, **we always catch an exception in a method that is, logically, the one in charge of handling such an exception.**

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

```
public void f() {
    g();
}
```

```
public void g() {
    safeCharAt("Hello, world!", -1);
}
```

```
public static char safeCharAt(String s, int i) {
    try {
        return s.charAt(i);
    } catch (IndexOutOfBoundsException e) {
        return '\n';
    }
}
```

```
public char charAt(int i) {
    // throws an exception
}
```


Ignoring checked (non-runtime) exceptions

- Let's get back to the first example:

```
public static void saveTextToFile(String filename, String text) {
        FileWriter writer = new FileWriter(filename);
        writer.write(text);
}
```

- Although we already saw how to catch the `IOException` thrown by the methods, let's say we decide to ignore them.
- **It will not compile.**
- As we saw, ignoring an exception rethrows it while making the calling method as the new throwing method. Java enforces that if we throw a **checked exception** we have to declare that in the method signature.
- This is what the new method signature would look like, if we decided to ignore the exception and let it be rethrown to the calling method:

```
public static void saveTextToFile(String filename, String text) throws IOException {
```


Throwing an exception manually

- Sometimes we realize, in our own method, that something went wrong, which we can't handle.
- If we want the caller method to handle this issue, we throw an exception.
- To throw an exception, we must follow these instructions:
  - Find an appropriate exception class, or write a custom exception class of our own.
  - Create a new exception object with the keyword '*new*'.
  - Use the non-default exception constructors to provide information about the issue.
  - Throw the reference to that exception object with the keyword '*throw*'.
  - If it's a checked exception, change the method signature to include the keyword '*throws*'.
  - Javadoc the exception in the method documentation.
- Example:
  - `throw new IllegalArgumentException("Invalid parameter value " + someParam);`

Example, throwing a runtime exception

```
/**
 * Returns a specific character in a specific string, capitalized.
 * @param s the non-blank string which includes the character.
 * @param i the non-negative index of the character within the string.
 * @return the character located at index i within string s, capitalized.
 * @throws IllegalArgumentException if s is null or blank.
 * @throws IllegalArgumentException if i is outside the string boundaries.
 */
public static char charAtCapital(String s, int i) {

        // validate parameters
        if (s == null || s.trim().length() == 0) {
                throw new IllegalArgumentException("String is null or empty");
        }
        if (i < 0 || i >= s.length()) {
                throw new IllegalArgumentException("Illegal index " + i);
        }

        return Character.toUpperCase(s.charAt(i));
}
```

Example, throwing a checked exception

- TBD

