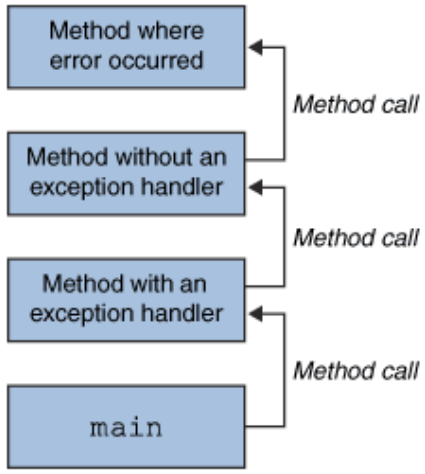
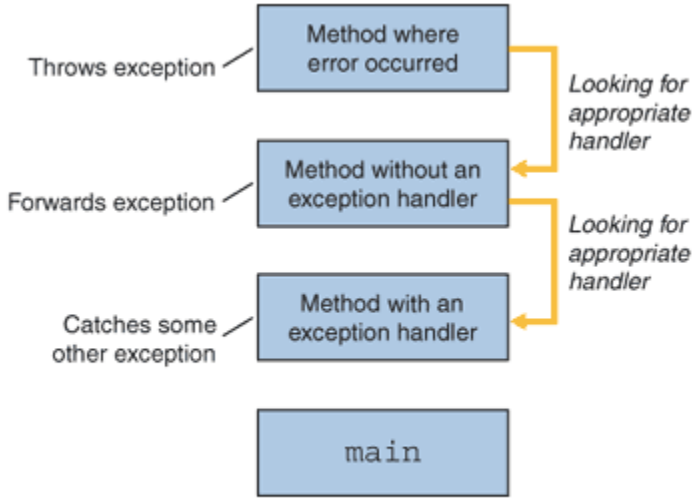
A. Modelul de tratare si propagare a exceptiilor

Poza 1: Stiva de apeluri



The call stack.

Poza 2: Cautarea in stiva de apeluri a metodei de tratare a exceptiei

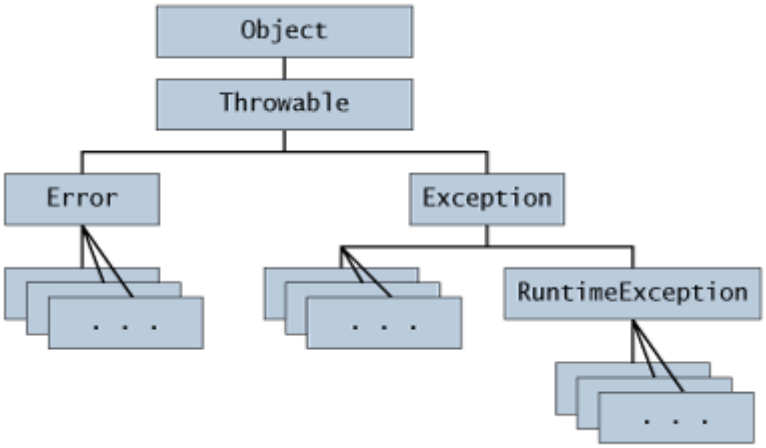


Searching the call stack for the exception handler.

B. Tipuri de exceptii:

1. *checked exception* (exceptii verificate) = NU trec de compilare. Se poate prevede mecanism de "recovery". Musai mecanism try-catch.
2. *errors* (erori) = trece de compilare DAR nu se poate prevede functionare defectuasa (e fizic stricat hard-diskul si la deschiderea de fisier se arunca 'java.io.IOError'). De obicei nu exista mecanism de try-catch.
3. *runtime exception* (exceptii la rulare) = trec de compilare DAR din cauza logicii de la dezvoltare defectuase rezulta din calcule numitor=0 si mai departe o impartire la 0. Se poate utiliza try-catch, dar mai bine se elimina bug-ul de reuseste ca din calcule sa rezulte numitor=0.

- 2+3 = *unchecked exception*



C. Crearea unei clase proprii care extinde clasa 'Exception'
D. Avantajele utilizarii exceptiilor

Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program. In traditional programming, error detection, reporting, and handling often lead to confusing spaghetti code. For example, consider the pseudocode method here that reads an entire file into memory.

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

At first glance, this function seems simple enough, but it ignores all the following potential errors.

- What happens if the file can't be opened?
- What happens if the length of the file can't be determined?
- What happens if enough memory can't be allocated?
- What happens if the read fails?
- What happens if the file can't be closed?

To handle such cases, the `readFile` function must have more code to do error detection, reporting, and handling. Here is an example of what the function might look like.

```
errorCodeType readFile {
    initialize errorCode = 0;

    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

Exceptions enable you to write the main flow of your code and to deal with the exceptional cases elsewhere. If the readFile function used exceptions instead of traditional error-management techniques, it would look more like the following.

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
       doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Note that exceptions don't spare you the effort of doing the work of detecting, reporting, and handling errors, but they do help you organize the work more effectively.