7. Exceptions
7. Exceptions

- Provide a **structured** form of **jump**
- **Exit** a group of nested **blocks** or function calls
- Data can be passed with the jump

An exception mechanism consists of two parts:
1. A construct to **raise** (or throw) an exception
2. A construct to **handle** (or catch) an exception

Raising an exception terminates the current computation and transfers control to the closest enclosing **exception handler** that can handle this kind of exception.
void work() {
    wake up;
    breakfast;
    catch bus;
    { teaching;
      research;
    };
    go home;
};

void work() {
    try {
        wake up;
        breakfast;
    } catch Late { run for bus }
    try {
        catch bus;
    } catch MissedBus {
        beg my wife for a ride }
    try {
        { teaching;
          research;
    }
    go home;
    } catch GoodWeather {
        go home and mow lawn };
};
Raising Exceptions

Exceptions can be raised implicitly by built-in functions in error situations or explicitly by the programmer through a `raise` construct.

```plaintext
{ exception WrongFormat;
  string f(string s) {
      if substr(s,1,7) = "<--XML" then
          ...
      else
          raise WrongFormat
  }
};
string in;
try {
      in := read("data");
      write(open("result"),f(in));
} catch ...

...;
```
Handling Exceptions

Exceptions that are raised inside of a try block can be handled through a series of `catch` constructs, i.e., the exception can be inspected and possible arguments can be bound.

```java
{ exception WrongFormat;
  string f(string s) { ... };
  string in;
  try {
    in := read("data");
    write(open "result",f(in));
  } catch (FileNotFoundException n) {
    print("File not found: "+n);
    write(open("result"),"error");
  } catch WrongFormat { 
    print("Incorrect format");
  }
  ...
};
```

- handle exceptions from this block
- test for specific exceptions
- bind argument
- continue "normal" computation
What Happens to the Runtime Stack?

```plaintext
{ exception WrongFormat;
    string f(string s) { ... };
    string in;
    try {
        in := read("data");
        write(open("result"), f(in));
    } catch (FileNotFoundException f) {
        print("File not found: "+f);
        write(open("result"), "error");
    }
}

...  

string read (string n) {
    ...  
    raise FileNotFoundException n 
    ...  
};
```

try/catch

read

try/catch

pop activation records immediately
Skipping Multiple Activation Records

```c
{ exception WrongFormat;
    string f(string s) { ... };
    string in;
    try {
        in := read("data");
        write(open("result"),f(in));
    } catch (FileNotFoundException f) { ... }
    catch WrongFormat { 
        print("Incorrect format");
    } ...
}
```

```c
string write (file p,string x) { ... };
```

```c
string f(string s) {
    raise WrongFormat 
};```

```c
try/catch
[⟨f=... in=?⟩]
[⟨F{} W{}⟩, ⟨f=... in=?⟩]
[⟨F{} W{}⟩, ⟨f=... in=\"a\"\⟩]
```

```c
write
[⟨W{}⟩, ⟨f=... in=\"a\"\⟩]
[⟨f=... in=\"a\"⟩]
```

```c
try/catch
[⟨p=* x=?⟩, ⟨F{} W{}⟩, ⟨f=... in=\"a\"⟩]
```

```c
f
[⟨s="a"⟩, ⟨p=* x=?⟩, ⟨F{} W{}⟩, ⟨f=... in=\"a\"⟩]
```

How to Find Handlers?

{ exception E;
    int x;
    { try {
        void f() { raise E }
        try {
            try {
                f();
            } catch E { x := 2; }
            } catch E { x := 3; }
            } catch E { x := 1; }
    } catch E { x := 2; }
    } catch E { x := 3; }
    } catch E { x := 1; }
};

void f() { raise E }

[⟨x=2⟩]
[⟨E{x:=1} f=...⟩, ⟨x=2⟩]

[⟨E{x:=3}⟩, ⟨E{x:=1} f=...⟩, ⟨x=2⟩]
[⟨E{x:=2}⟩, ⟨E{x:=3}⟩, ⟨E{x:=1} f=...⟩, ⟨x=2⟩]

[⟨E{x:=2}⟩, ⟨E{x:=3}⟩, ⟨E{x:=1} f=...⟩, ⟨x=2⟩]

[⟨x=2⟩]
[⟨E{x:=1} f=...⟩, ⟨x=2⟩]

[⟨E{x:=3}⟩, ⟨E{x:=1} f=...⟩, ⟨x=2⟩]
[⟨E{x:=2}⟩, ⟨E{x:=3}⟩, ⟨E{x:=1} f=...⟩, ⟨x=2⟩]

dynamic scoping for exception handlers!
Design Rule for Using Exceptions

Whenever a function/method/procedure knows that an error situation exists, but cannot know how to deal with this situation in general, an exception carrying useful information about the error should be raised.

Why can it happen that a function does not know how to deal with an error directly? Because proper error treatment may depend on the usage context.

All function calls that might raise an exception should be eventually wrapped by a handler for all possible exceptions.
Exceptions in Java

Exceptions in Java are represented by objects, which are instances of the class Throwable or any of its subclasses.

Nice aspects of "exceptions as objects":

1. Objects can carry information about exceptions (in fields)
2. Subtyping can be exploited in handling exceptions: a handler for an exception of class E can deal with exceptions of any of E's superclasses.
Try/Catch Block

```java
try {
    File f = new File("data");
    process(f);
    f.close();
} catch (FileOpenFailed e) {
    System.err.println("Cannot open: " + e);
    e.printStackTrace();
}
```

- **try** { try block might throw exception(s)
- **catch** (FileOpenFailed e) { a subclass of Throwable variable bound to exception object
- only FileOpenFailed exceptions thrown in the **try** block are caught by this **catch** clause
Finally Block

```java
try {
    File f = new File("data");
    process(f);
    f.close();
} catch (FileOpenFailed e) {
    System.err.println("Cannot open: "+ e);
    e.printStackTrace();
} finally {
    System.out.println("The End");
}
```

the **finally** block will always be executed, no matter whether or not an exception occurred, no matter whether or not the exception was caught.
Syntax

try blocks can be nested
each try block can have multiple catch blocks
the finally block is optional

User-defined exceptions

User-defined exceptions are realized through subclass definitions
Thrown exceptions that are from a subclass of Exception must be enclosed by a catch block for that exception
is enforced by the compiler!