6. Exceptions
6. 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.
Exceptions for Exceptional Situations

```java
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 ...
  ...
};
```

declaring an exception
"normal" computation
user-raised exception
system-raised exceptions
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.

```plaintext
{ 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");
  }
}
```
What Happens to the Runtime Stack?

```java
{ 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
    ...
};
```

- **Exception Handling**: The code handles exceptions by defining an exception `WrongFormat` and catching it with a `FileNotFoundException`.
- **Runtime Stack**: The diagram illustrates how the runtime stack changes during exception handling. The stack pops activation records immediately when the `catch` block is executed.
- **try/catch Blocks**: The `try` block reads a file and writes the result, while the `catch` block prints an error message if the file is not found.

The diagram shows how the stack frames are pushed and popped as the code executes and handles exceptions.
Exceptions

Skipping Multiple Activation Records

```java
{ 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");
    } ...
}

catch (FileNotFoundException f) { ... }

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

string f(string s){ 
    raise WrongFormat 
};
```
How to Find Handlers?

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

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();
}
```

might throw exception(s) a subclass of Throwable

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
More ...

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!