INTRODUCTION TO MODSIM II
MODSIM II IS

- A general-purpose language
  - Strong data typing
  - Easy modularisation
  - high-level

- An Object-Oriented language
  - Supports encapsulation, data abstraction, inheritance
    multiple inheritance and polymorphism

- A simulation language
  - Supports the process view of simulation
  - Enables interactive graphics and animation
MODSIM II INCLUDES

- **mscomp, the smart compiler**
  - Generate 'C' code for portability
  - Automatically handles module dependencies (no 'make'-files)

- **Extensive set of libraries**
  - Providing resources, queues, statistics, streamed I/O etc.
  - Supports graphics and simulation

- **Simdraw, the graphics editor**
  - Uses windowing environment, such as X-windows
  - Creates icons, menus, dialogue boxes and presentation graphics

- **Simvideo, the playback utility**
  - Permits playback and editing of graphics sessions
  - Useful for presentation purposes
1. MODSIM II PROGRAMMING LANGUAGE

1.1 Identifiers and Simple Data Types

1.2 Structured Data Types

1.3 Control Structures

1.4 Procedures

1.5 Program Layout and Modules

1.6 Built-In Procedures & Some Useful Libraries
1. MODSIM II PROGRAMMING LANGUAGE

1.1 Identifiers and Simple Data Types
IDENTIFIERS

- Names of user-defined constructs, such as modules, constants, variables, types, procedures.

- Must begin with an alphabetic character.

- Can be followed by any number of alphabetic or numeric characters.

- Note: MODSIM II is case sensitive!!

examples:

<table>
<thead>
<tr>
<th>Good Examples</th>
<th>Bad Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoOfTrucks,</td>
<td>=&gt; ✔️</td>
</tr>
<tr>
<td>A32, a32,</td>
<td></td>
</tr>
<tr>
<td>count, Count</td>
<td>=&gt; ☠️  ⚡️</td>
</tr>
<tr>
<td>number_of_Trucks,</td>
<td>=&gt; ☠️  ⚡️</td>
</tr>
<tr>
<td>A.32, 32a, $count, Count%</td>
<td></td>
</tr>
</tbody>
</table>
**SIMPLE DATA TYPES**

- **INTEGER**  
  exact representation of each whole number  
  Range: -2,147,483,648 to 2,147,483,647  
  examples: 22 0E4 -259

- **REAL**  
  fractional or floating point numbers  
  Range: 1.7*10^-308 to 1.7*10^308  
  examples: 23.45 34.2E12 0.0 -13.456453

- **CHAR**  
  single character  
  examples: 'A' 'a' '$' '%' ''

- **STRING**  
  sequence of characters  
  examples: "Welcome" "Nice Day"

- **BOOLEAN**  
  TRUE or FALSE state
CONSTANT DECLARATIONS

- Define an expression which is substituted at compile time for the specified identifier wherever it occurs in the program

example:

```plaintext
CONST

pi = 3.141592654;
circ = 2.0 * pi;
twopi = circ;
hello = "Hello";
```
**TYPE DECLARATIONS**

- Used to create new data types which build upon existing types or user-defined types.

- Enumerated types are an ordered collection of values expressed as identifiers. They are enclosed in curved brackets ( ).

- Subrange types are any subset of an ordinal type. They are enclosed in square brackets [ ].

**examples:**

```
TYPE
  DirectType = (up, down);
  DayType    = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
  ScoreType  = [1..10];
  GradeType  = ['A'..'F'];
```
**VARIABLE DECLARATIONS**

- Reserves memory space to hold the data.

- Assigns names to these data storage areas.

- Can use any user-defined types.

examples:

```plaintext
VAR
    n, j, k : INTEGER;
    x, y    : REAL;
    filename : STRING;
    day      : DayType;
    direction : DirectType;
```
AUTOMATIC INITIALISATION OF VARIABLES

- Each variable is automatically initialised to a specific value at the time it is declared:

  - INTEGER: 0
  - REAL: 0.0
  - BOOLEAN: FALSE
  - STRING: null string
  - CHAR: CHR(0)
  - enumerated types: first item in list
Example of Simple Data Types

Main Module simpledt;

Const
  StdPktSize = 1024;
  VoicePkt = 'V';
  DataPkt = 'D';

Type
  PktPri = (Deferred, Routine, Priority, Immediate, Flash);

Var
  source : STRING;
  size : INTEGER;
  startTime : REAL;
  type : CHAR;
  status : BOOLEAN;
  priority : PktPri;

Begin
  source := "Space Station";
  size := StdPktSize;
  startTime := 0.0;
  type := VoicePkt;
  status := TRUE;
  priority := Priority;

End Module.
1. MODSIM II PROGRAMMING LANGUAGE

1.2 Structured Data Types
STRUCTURED DATA TYPES

- ARRAY
  Ordered set of data elements referenced by an index where all elements are of the same type (called the base type). The index has to be an ordinal type. May be fixed or dynamic.

- RECORD
  User-defined data structure of other data structures or built-in types. Each element is a named field which can be referenced individually. May be fixed or dynamic.
**FIXED ARRAYS**

- All elements contained in the array must be of fixed size, i.e. known at compile time.

- Can have many dimensions.

Example:

```pascal
TYPE
   DayType   = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
   DailyCnt  = FIXED ARRAY DayType OF INTEGER;
   Square    = (blank, x, o);
   TicTacType = FIXED ARRAY [1..3], [1..3] OF Square;

VAR
   ValList : FIXED ARRAY [-2..10] OF REAL;
   TicTac  : TicTacType;
   x      : REAL;
   y, z   : Square;

BEGIN
   x := ValList[-2];
   y := TicTac[3][1];
   z := TicTac[1,1];
```
DYNAMIC ARRAYS

- Similar to fixed arrays, but here the elements can be of variable size.

- Memory must be explicitly allocated / de-allocated using the built-in procedures NEW and DISPOSE.

- Null value for an array variable is NILARRAY.

eexample:

```pascal
VAR
    ValList : ARRAY INTEGER OF REAL;
    n : INTEGER;
BEGIN
    n := ... some calculation;
    NEW(ValList, 1..n);
    DISPOSE(ValList);
    ...
```
**FIXED RECORDS**

- Each element is called a field and must be of fixed size.

- Individual fields are referenced by appending a period (.) to the record identifier and then specifying the field name.

**example:**

```plaintext
TYPE
    CharArray = FIXED ARRAY [1..3] OF CHAR;
    Packet = FIXED RECORD
        source : CharArray; (* note: FIXED ARRAY! *)
        size : INTEGER;
        startTime : REAL;
        type : CHAR;
    END RECORD;

VAR
    IsoPkt : Packet;

BEGIN
    IsoPkt.size := 1024;
```
**Dynamic Record**

- Similar to fixed record, but here fields are allowed to be of variable size.
- Memory is explicitly allocated / de-allocated using `NEW` and `DISPOSE`.
- The null value for a record is `NILREC`.

**Example:**

```
TYPE
  Packet = RECORD
    source : STRING; (* NOTE: variable Size!! *)
    size : INTEGER;
  END RECORD;

VAR
  IsoPkt : Packet;
BEGIN
  NEW(IsoPkt);
  IsoPkt.source := "SpaceStation";
  DISPOSE(IsoPkt);
```
1. MODSIM II PROGRAMMING LANGUAGE

1.3 Control Structures
## OPERATORS

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>:=</td>
<td></td>
</tr>
<tr>
<td>addition</td>
<td>+</td>
<td>INTEGER, REAL</td>
</tr>
<tr>
<td>subtraction</td>
<td>−</td>
<td>INTEGER, REAL</td>
</tr>
<tr>
<td>multiplication</td>
<td>*</td>
<td>INTEGER, REAL</td>
</tr>
<tr>
<td>real division</td>
<td>/</td>
<td>REAL</td>
</tr>
<tr>
<td>integer division</td>
<td>DIV</td>
<td>INTEGER</td>
</tr>
<tr>
<td>modulus</td>
<td>MOD</td>
<td>INTEGER</td>
</tr>
<tr>
<td>unary plus</td>
<td>+</td>
<td>INTEGER, REAL</td>
</tr>
<tr>
<td>unary minus</td>
<td>−</td>
<td>INTEGER, REAL</td>
</tr>
<tr>
<td>string concatenate</td>
<td>+</td>
<td>STRING</td>
</tr>
</tbody>
</table>

- Use the built-in procedures to perform type conversions, e.g. from INTEGER to REAL use FLOAT (INTEGER).
WHEN ARE TYPES COMPATIBLE?

- Both types are the same:
  
  \[
  \begin{align*}
  a & : \text{INTEGER}; \\
  b & : \text{INTEGER}; \\
  \end{align*}
  \]

- They are explicitly defined to be equal in type:
  
  \[
  \text{TYPE} \\
  \quad \text{type1} = \text{type2}; \\
  \]

- One is a subrange of another:
  
  \[
  \text{TYPE} \\
  \quad \text{DayType} = (\text{Sun, Mon, Tue, Wed, Thu, Fri, Sat}); \\
  \quad \text{WeekDay} = (\text{Mon, Tue, Wed, Thu, Fri}); \\
  \]

- Both are subranges of the same type:
  
  \[
  \text{TYPE} \\
  \quad \text{YearDayNumType} = [1..365]; \\
  \quad \text{WeekDayNumType} = [1..7]; \\
  \]

- \text{CHAR} is compatible with \text{STRING}, but not vice versa!!
RELATIONAL OPERATORS

- Compare values.
- All binary operators.
- The operands must be of the same type.
- The result is BOOLEAN.

\[
\begin{align*}
= & \quad \text{equal} \\
<> & \quad \text{not equal} \\
< & \quad \text{less than} \\
\leq & \quad \text{less than or equal} \\
> & \quad \text{greater than} \\
\geq & \quad \text{greater than or equal}
\end{align*}
\]
LOGICAL OPERATORS

AND  OR  NOT

- AND, OR take two operators.
- NOT takes one operator.
- The result is BOOLEAN.
- MODSIM II uses 'short-circuit' evaluation of boolean expressions.

example:

\[
\text{IF } (x<>0.0) \text{ AND } (y/x<20.0) \ldots
\]

if \( x = 0.0 \): \( (x<>0.0) \) evaluates to FALSE so \( (y/x<20.0) \) is no longer evaluated, hence no runtime error
IF - BRANCHING

IF month <= 3
    quarter := "1st quarter";
ELSIF month <= 6
    quarter := "2nd quarter";
ELSIF month <= 9
    quarter := "3rd quarter";
ELSE
    quarter := "4th quarter";
END IF;

- The ELSIF-clauses are optional (note their spelling)!

- Any number of statements can follow a clause (no need for BEGIN .. END block).
CASE - BRANCHING

CASE month
  WHEN 1..3:  quarter := "1st quarter";
  WHEN 4, 5, 6:  quarter := "2nd quarter";
  WHEN 7..9:  quarter := "3rd quarter";
  OTHERWISE  quarter := "4th quarter";
END CASE;

- The OTHERWISE-clause is optional, but if it is omitted, then one of the WHEN-clauses must evaluate, otherwise a run-time error occurs!
WHILE - LOOP

- Loop which is repeated 0 or more times.

- As long as the boolean expression at the head of the WHILE construct remains true, enclosed statements are executed.

example:

```plaintext
n := 2;
WHILE n < 5
    OUTPUT("n = ", n);
    n := n + 1;
END WHILE;

=> n = 2
    n = 3
    n = 4
```
REPEAT - LOOP

- Loop which is repeated 1 or more times.

- Boolean expression at the end of the statement is not evaluated until the body of the loop has been executed at least once.

example:

```plaintext
REPEAT
    OUTPUT("This will print at least once");
    INC(a);
UNTIL (a > 5);
```
FOR - LOOP

- Increments / decrements a loop variable by an integral value through a specified range, each time repeating the enclosed statements.

- The loop variable may be of any ordinal type.

- An additional `BY` statement may be added. The stepping value must be compatible with type `INTEGER`.

example:

```plaintext
FOR n := 1 TO 5 (* BY 1 *)
    OUTPUT("The next number is ", n);
END FOR;

FOR n := 5 DOWNTO 1 (* BY 1 *)
    OUTPUT("The next number is ", n);
END FOR;
```
**INFINITE LOOP**

- **LOOP** repeats the enclosed statements forever.

- **EXIT**-statement can be used to break out of the loop.

- **Note:** **EXIT** can be used to break out of any kind of loop!!

**example:**

```plaintext
LOOP
    OUTPUT("Beginning loop value of IntVar is ", IntVar);
    INC(IntVar);
    IF IntVar > 5
        EXIT;
    END IF;
END LOOP;
OUTPUT("Ending loop value of IntVar is ", IntVar);
```
FOR EACH LOOP

- This will be discussed in the section on Grouping Objects.
STANDARD INPUT / OUTPUT

- Standard input from the keyboard can be obtained by using the reserved word INPUT.

- Standard output to the terminal can be obtained by using the reserved word OUTPUT.

example:

```plaintext
MAIN MODULE hello;

VAR
    FirstName : STRING;
BEGIN
    OUTPUT("Please enter your first name:");
    INPUT(FirstName);
    OUTPUT;
    OUTPUT("Hello, ", FirstName, " !");
END MODULE.
```
SINGLE-KEYSTROKE INPUT

- **Input** requires the user to hit the return key before the data is read in. If you try to read a character, hitting return will read the input stream from the keyboard and the CR will be stored instead of the single character you try to read in.

- To read single characters, use the procedure `ReadKey()` declared in the library module `IOMod`.

example:

```plaintext
FROM IOMod IMPORT ReadKey;
VAR
  char : CHAR;
...
BEGIN
  char := ReadKey();
```
FORMATTED STANDARD OUTPUT

- Produce formatted output to the terminal using the PRINT WITH statement.

examples:

```
BEGIN
  r := 32.854;
  i := 28;
  FirstName := "486 Personal Computer";
  PRINT (FirstName, r, i) WITH
    "I', a *****************. r=***.* i=***.";
END MODULE.
```

=> I'm a 486 Personal Computer. r=32.9 i=28.

- Note: with real numbers the fractional part is rounded to match the number of decimal fields that are specified in the PRINT . . WITH statement.
FORMATTED STANDARD OUTPUT (CONT.)

- If the specified field is too short, just the *'s are printed.

- If the specified field is too long, the unused places are filled by spaces.

- Use <, > and ~ to justify the fields left, right and centred respectively. But note: <, > and ~ are themselves interpreted as places! (so '***.*< ' gives 2dp).

- The built-in procedure SPRINT..WITH performs the same function but in addition returns the result as a string. It must be used in form of an assignment to a variable of type STRING.

  e.g:    myString := SPRINT (.....) WITH ".........";
1. MODSIM II PROGRAMMING LANGUAGE

1.4 Procedures
**PROCEDURES**

- Procedures are named blocks of code which may be invoked from other parts of the program by stating the procedure name and providing the required parameters.

- MODSIM II distinguishes between proper and function procedures.

- Both kinds of procedures may have a formal parameter list for passing information in and out.

- Only difference between the two kinds is the way in which they are invoked and declared.
FUNCTION PROCEDURES

example:

PROCEDURE Mean(IN AnArray : SomeArrayType): REAL;
BEGIN
  ... 
  RETURN(...);
END PROCEDURE;
...

x := Mean(AnArray);

- Declaration must specify the return type (indicated by ':REAL' above).
- Must be called right of an assignment operator with the procedure name.
- Must have a RETURN-statement.
- If no arguments are passed, then must indicate the empty brackets ( ).
**PROPER PROCEDURES**

example:

```delphi
PROCEDURE Mean(IN AnArray : SomeArrayType; OUT y : REAL);
BEGIN
    ...
    END PROCEDURE;
    ...
    Mean(x, y);
```

- Return variables must be specified inside the parameter list (here 'OUT y : REAL').

- Must be called by a single statement of the procedure name with its parameters.

- If no arguments are passed, then the brackets () can be omitted.

- No return statement must be included in the body of the procedure.
FORMAL PARAMETER QUALIFIERS

**IN**  
Value may only be passed to the procedure from caller (pass by value). A copy is passed to the procedure, except for **FIXED ARRAY**.

**INOUT**  
Value may be passed in either direction (pass by reference). No copy is made. The formal parameter is simply an alias for the actual parameter. The variable has to be initialised.

**OUT**  
Operates similar to **INOUT**, but here the variables are initialised as they are passed in. Information flows in only one direction.
FORWARD DECLARATION

• In order to circumvent the 'declaration before use' rule, declare the procedure as **FORWARD**.

• But: the parameter list specified in the declaration must match the parameter list in the implementation part.

• Can be used for example to type the procedures in alphabetical order.

example:

```
PROCEDURE DoSomething(IN x : INTEGER); FORWARD;
```
1. MODSIM II PROGRAMMING LANGUAGE

1.5 Program Layout & Modules
PROGRAM LAYOUT

- Every program must have a **MAIN MODULE**.

- The **MAIN MODULE** consists of two main parts: declaration and implementation.

- In the declaration part, the constants, types, variables and procedures are defined.

- In the implementation part, the executable code is specified.
PROGRAM LAYOUT EXAMPLE

MAIN MODULE somename;

(* Declaration Part *)   (* note: this is a remark *)
CONST

    constant declarations

TYPE

    type declarations

VAR

    variable declarations

PROCEDURE x;
BEGIN
    ...
    procedure implementation
END PROCEDURE;

PROCEDURE y (...):z;
BEGIN
    ...
    procedure implementation
END PROCEDURE;

{ Implementation Part }   (* note: this is another remark *)
BEGIN
    ...
END MODULE.
Blocks and Scope

- A block is a structural unit of a MODSIM II program.

- A block consists of constant, type, variable and procedure declarations, followed by executable code.

- Identifiers which name constants, types, variables and procedures in a block are known or 'visible' only within that block. We say their scope is limited to that block.
NESTING OF BLOCKS

MAIN MODULE sample3;

VAR
    x, y: REAL;
PROCEDURE foo;
VAR
    a, b: REAL;
BEGIN
    ...
END PROCEDURE;
BEGIN (* main program starts here *)
    ...
    x := 12.34;
    OUTPUT("first : ", x);
    first : 12.34
END BLOCK
VAR
    x, b, z: INTEGER;
BEGIN
    x := 5;
    OUTPUT("second : ", x);
    second : 5
END BLOCK;
OUTPUT("third : ", x);
third : 12.34
...

END MODULE.
LIBRARY MODULES

- A MODSIM II program can have any number of library modules.

- Each library module is separated into two sub-modules, stored in different files: DEFINITION and IMPLEMENTATION MODULE.

- The DEFINITION MODULE contains all the declarations to the corresponding IMPLEMENTATION MODULE (marked as (*declaration part *) in the example above).

- The IMPLEMENTATION MODULE contains the executable code (marked as the {implementation part} in the example above).
LIBRARY MODULES: EXAMPLE

DEFINITION MODULE somename;
   CONST
     constant declarations
   TYPE
     type declarations
   VAR
     variable declarations
     PROCEDURE a;
     PROCEDURE b (...): c;
END MODULE.

IMPLEMENTATION MODULE somename;
  PROCEDURE a;
  BEGIN
   .
   END PROCEDURE;
  PROCEDURE b (...): c;
  .
  END PROCEDURE;
END MODULE.
**Naming Conventions for Modules**

- Main Module must have file-name starting with 'M' and ending with '.MOD'.

- Definition Module must have a file-name starting with 'D' and ending with '.MOD'.

- Implementation Module must have a file-name starting with 'I' and ending with '.MOD'.

- Definition and Implementation Modules must have the same file name, apart from the extension

example:

```
MHello.Mod  => Main Module
DPart1.Mod  => Definition Module
IPart1.Mod  => Implementation Module
```
COMBINING MODULES

- Data types declared in other library modules may be used by using the `FROM ... IMPORT` statement.

- These must be the first statements in any module, immediately following the Module name.

- Anything declared in a Definition Module is automatically visible in the associated Implementation Module and hence need not be imported.

- Dependencies between modules may be cyclical.

examples:

```
FROM GTypes IMPORT ALL ColorType; => all colours
FROM GTypes IMPORT ColorType(Red, Blue); => subset
FROM GTypes IMPORT ColorType (Blue AS bl, Red AS r);
```
**Module Initialisation**

- **MAIN** and **IMPLEMENTATION** Modules may contain a procedure called **ModInit**.

- **ModInit** will be automatically executed before anything else in the module. It may not be passed any arguments and cannot return any values.

- Where more than one **ModInit** exists in a program, no guarantee can be made about their order of execution.
A NOTE ON DEBUGGING

• The keyword `TRACE` provides information about procedure / method executions at the time when the command is executed.

• In case of a run time error, the traceback mechanism will automatically print out procedure / method name, the line in which the error occurred and any local variables and arguments. (Remember to set the traceback flag when compiling!!)

• Facilities to track memory allocations are provided in the library module `Debug`.

• Note: release 1.9 of Modsim II comes with a full portable debugger.
1. MODSIM II PROGRAMMING LANGUAGE

1.6 Built-In Procedures & Some Useful Libraries
BUILT-IN PROCEDURES

- Can be used without declaration or importing.

ABS(IN arg: INTEGER or REAL): INTEGER or REAL;
CAP(IN ch: CHAR): CHAR;
CHARTOSTR(IN chrArray : ARRAY OF CHAR): STRING;
CHR(IN n: INTEGER): CHAR;
DEC(INOUT ard: AnyOrdinalType []; IN n: INTEGER]);
DISPOSE(IN refVar : AnyRefType);
FLOAT(IN n: INTEGER): REAL;
HALT;
HIGH(IN arr: AnyArrayType): IndexType;
INC(INOUT arg: AnyOrdinalType []; IN n: INTEGER]);
INTTOSTR(IN n: INTEGER): STRING;
LOW(IN arr: AnyArrayType): IndexType;
LOWER(IN str: STRING): STRING;
MAX(ScalarType): ScalarType;
MIN(ScalarType): ScalarType;
NEW(OUT rec: AnyRecordType);
BUILT-IN PROCEDURES (CONT.)

NEW(OUT array: AnyArrayType);
ODD(IN n: INTEGER): BOOLEAN;
ONERROR(IN proc: AnyProcedure);
ONEXIT(IN proc: AnyProcedure);
ORD(IN arg: AnyOrdinalType): INTEGER;
REALTOSTR(IN x: REAL): STRING;
ROUND(IN arg: REAL): INTEGER;
SIZEOF(AnyTypeName): INTEGER;
STRLEN(IN str: STRING): INTEGER;
STRTOCHAR(IN str: STRING; OUT chrArray: ARRAY INTEGER OF CHAR);
STRTOINT(IN str: STRING): INTEGER;
STRTOREAL(IN str: STRING): REAL;
UPPER(IN str: STRING): STRING;
TRUNC(IN arg: REAL): INTEGER;
**Math Module MathMod**

- Provides mathematical functions.

```plaintext
PROCEDURE SIN(IN x: REAL): REAL;
PROCEDURE COS(IN x: REAL): REAL;
PROCEDURE TAN(IN x: REAL): REAL;
PROCEDURE SQRT(IN x: REAL): REAL;
PROCEDURE EXP(IN x: REAL): REAL;
PROCEDURE LN(IN x: REAL): REAL;
PROCEDURE LOG(IN x: REAL): REAL;
PROCEDURE POWER(IN x, y: REAL): REAL;
PROCEDURE CEIL(IN x: REAL): INTEGER;
PROCEDURE FLOOR(IN x: REAL): INTEGER;
pi = 3.1415926535897932;
e = 2.7182818284590452;
```
UTILITY MODULE UTILMOD

- Provides useful utilities for interface with the operating system.

PROCEDURE DateTime(OUT DT: STRING);
PROCEDURE ClockTimeSecs(): INTEGER;
PROCEDURE Delay(IN NumSeconds: INTEGER);
PROCEDURE ExitToOS(IN Status: INTEGER);
PROCEDURE GetNumArgs(): INTEGER;
PROCEDURE GetCmdLineArg(IN ArgNumber: INTEGER; OUT Arg: STRING);

- Note: argument number 0 is considered to be the program name!
DEBUGGING MODULE DEBUG

- Provides simple debugging facilities.

PROCEDURE GetNumberType(IN typeid: INTEGER): INTEGER;
PROCEDURE GetNumberArrays(): INTEGER;
PROCEDURE GetNumberStrings(): INTEGER;
PROCEDURE PrintMemStats(IN stream: StreamObj);
PROCEDURE WriteTrace(IN filename: STRING);
CONTROL MODULE CRTMod

- Provides a procedure to clear the screen;

PROCEDURE ClearScreen;
2. OBJECT - ORIENTED PROGRAMMING

2.1 Objects in MODSIM II

2.2 Advantages of Object - Orientation

2.3 Advanced Object - Oriented Features

2.4 Grouping of Objects

2.5 More Useful Libraries
2. OBJECT - ORIENTED PROGRAMMING

2.1 Objects in MODSIM II
TRADITIONAL VS OBJECT-ORIENTED WELTANSCHAUUNG

• Traditional 'imperative' approach to software:
  - A single process (program) acts upon a database.
  - The process behaves differently according to the state of the data.
  - Design is data driven.

• Object-Oriented approach to software:
  - Data and the processes which change them are encapsulated into objects.
  - Objects communicate through messages.
  - Design is object-driven and is a natural progression from the process of modelling the problem domain.
OBJECTS IN MODSIM II

- An Object is the encapsulation of data (fields) and procedures (methods).

- Fields define an object's state at any instant. They can only be modified by the object in which they are contained, but they can be read from any part of the program.

- Methods describe the actions which the object can perform. Only an object's methods can change the fields.

- Objects are dynamic data types, like RECORDs. They can be created and destroyed by using NEW and DISPOSE.
DEFINING AN OBJECT

- Objects are declared as type `OBJECT` in the `TYPE` declaration section of the module.

- The type declaration defines the fields and methods, but contains no single line of executable code.

- The null value of an object variable is `NILOBJ`.

Example:

```pascal
TYPE
   ComplexNbrObj = OBJECT
      RPart    : REAL;
      IPart    : REAL;
      ASK METHOD SetValue (IN rpart, ipart: REAL);
      ASK METHOD AddComplex (IN a, b: REAL);
   END OBJECT;

VAR
   x, y, z: ComplexNbrObj;
```
IMPLEMENTING AN OBJECT

- Usually follows the section on **PROCEDURE** implementations in a module, starting with the keyword **OBJECT** and the object type identifier.

- Object implementation provides the executable code for all the methods, similar to the implementation of procedures.

example:

```plaintext
OBJECT ComplexNbrObj;
   ASK METHOD SetValue (IN rpart, ipart: REAL);
   BEGIN
      ...
      END METHOD;

   ASK METHOD AddComplex (IN a, b: REAL);
   BEGIN
      ...
      END METHOD;

END OBJECT;
```
OBJECT TYPE & OBJECT INSTANCE

• The type declaration simply provides a description of the objects' characteristics. (Called 'Class' in other OO-programming languages).

• Particular objects of an object type have to be created / destroyed using NEW and DISPOSE. These are allocated in memory and called 'instances'.

• Each object instance has its own copy of the fields, and a reference to the methods pertaining to its type., (i.e. no copy of the methods is made for each individual instance, they all reference the same code).

examples:

    VAR
    x, y, z : ComplexNbrObj;
    BEGIN
    NEW(x);
    NEW(y); (* creates individual instances *)
    NEW(z);
SCOPE OF FIELDS AND VARIABLES

- Each field declared within an object type is visible to all its methods.

- The object instance itself has 'read' and 'write' access to its fields, other object instances only have 'read' access.

- Each instance has its own copy of the fields, but the methods are shared by all objects of the same type.

- Variables declared in a method implementation are local to that particular method. There is a unique copy of that local variable for each invocation of the method.
REFERENCING THE FIELDS

- To 'read' the field of an object instance, use the keyword ASK followed by the object instance identifier and the field name.

- Alternatively, use the dot '.' notation: separate the instance identifier and the field name by a dot.

- Fields cannot be on the left hand side of an assignment statement other than in one of the object's methods.

examples:

realvalue := ASK x.RPart;
imagvalue := ASK x.IPart;

realvalue := x.RPart;
imagvalue := x.IPart;
OBJECT METHODS

• Methods define the operations that can be performed on instances of an object type.

• They are like procedures or functions, but are always tightly bound to an object type.

• There are 3 kinds of methods:

  ASK METHODS
  TELL METHODS
  WAITFOR METHODS

• TELL METHODS and WAITFOR METHODS will be discussed later.
ASK METHODS

- **ASK METHODS** are executed synchronously, i.e. the statements in the method are executed as soon as the method is called (just like procedures).

- They may be proper or function methods:
  
  proper: \[\text{ASK METHOD} \ \text{SetValue}(\text{IN} \ rpart, \ ipart: \text{REAL})\];
  
  function: \[\text{ASK METHOD} \ \text{IsComplex}(\text{IN} \ a, \ b: \text{REAL}): \text{BOOLEAN}\];

- They may take **IN**, **OUT** or **INOUT** parameters.

- They are invoked as follows:

  \[\text{ASK} \ x \ \text{TO} \ \text{SetValue}(4.0, \ 5.0)\];
example:

MAIN MODULE Complex;
TYPE
  ComplexNbrObj = OBJECT
    RPart : REAL;
    IPart : REAL;
  ASK METHOD SetValue(IN rpart, ipart: REAL);
  ASK METHOD AddComplex(IN a, b: REAL);
END OBJECT;
VAR
  x, y : ComplexNbrObj;
OBJECT ComplexNbrObj;
  ASK METHOD SetValue(IN rpart, ipart: REAL);
  BEGIN
    RPart := rpart;
    IPart := ipart;
  END METHOD;
  ASK METHOD AddComplex(IN a, b: REAL);
  BEGIN
    RPart := RPart + a;
    IPart := IPart + b;
  END METHOD;
END OBJECT;
BEGIN (* MAIN *)
  NEW(X); NEW(y);
  ASK x TO SetValue(1.0, 3.0);
  ASK y TO SetValue(9.0, 7.0);
  ASK x TO AddComplex(y.RPart, y.IPart);
END MODULE.
SELF

- Used to refer to the object instance itself

```
Goto(Dest);
Position := Location();
```

```
ASK SELF TO Goto(Dest);
Position := ASK SELF Location();
```

- SELF may also be used when an object wants to identify itself to another object.

```
ASK Radar TO GiveWarningTo(SELF);
```
2. OBJECT-ORIENTED PROGRAMMING

2.2 Advantages of Object-Orientation
KEY FEATURES OF OBJECT-ORIENTATION

- **Encapsulation:** Data and the processes which act upon them are encapsulated within an object.

- **Message Passing:** Objects communicate via messages.

- **Inheritance:** Objects can inherit the capabilities of other objects.

- **Polymorphism:** Different object types can respond to the same message.

- **Data Hiding:** Data can be protected access for other objects.
ENCAPSULATION

• Only the methods of an object can alter the values of the data fields.

examples:

```
TYPE
  MammalObj = OBJECT
    weight: REAL;
    ASK METHOD SetWeight(IN rel: REAL);
  END OBJECT;

VAR
  Monkey:  MammalObj;
  OBJECT MammalObj;
  ASK METHOD SetWeight(IN x: REAL);
  BEGIN
    weight := x;
  END METHOD;
END OBJECT;

BEGIN
  NEW(Monkey);
  ASK Monkey TO SetWeight(10.0); (* not: Monkey.weight := 10.0 *)
  ...
```
ENCAPSULATION (CONT.)

- Encapsulation increases safety.

- An instance cannot accidentally affect another object's data.

- An instance only needs to know about the existence of a method. The actual implementation of that method is irrelevant (to other instances).

- The method implementation can incorporate consistency checks, so that any parameter value can be handled by the method - whether valid or invalid.

example:

A car-object has a method to increase the speed. The driver-object invokes this method and submits the new speed as a parameter. The implementation of the method can be such as to handle even speeds which are technically infeasible for a particular car type. Hence, the driver-object does not need to know all the details. All it needs to know is that the method to increase the speed exists and what parameters to give it.
MESSAGE PASSING

- The invocation of an `ASK` method is interpreted as 'sending a message'. (Similarly for the other types of methods discussed later).

- Message passing provides a formal interface specification:
  - the means by which objects communicate are clearly defined.
  - messages only specify the interface between the rest of the program and the method. The implementation of the method can change without being apparent to the 'outside world'.
**INHERITANCE**

- Objects can inherit the capabilities of other objects, i.e. take over their fields and methods.
- This increases reusability of models by building up libraries of objects.

![Inheritance Diagram](image)
INHERITED OBJECT TYPE DECLARATION

- To inherit the fields and methods from another object type, simply state the parent object's identifier in brackets ( ) after the keyword OBJECT.

example:

```plaintext
MammalObj = OBJECT
    weight : REAL;
    ASK METHOD Eat(IN foodWeight: REAL);
    ASK METHOD SetWeight(IN initWeight: REAL);
END OBJECT;

HumanObj = OBJECT(MammalObj)
    ASK METHOD Talk(IN word: STRING);
END OBJECT;
```

- MammalObj is the parent, HumanObj is a child of MammalObj.
MULTIPLE INHERITANCE

- An object type may be defined in terms of several parent object types.
- The child inherits all the fields and methods from all of its parents.
- Multiple inheritance is declared by listing each of the parents in the object type declaration.
MULTIPLE INHERITANCE (CONT.)

example:

FishObj = OBJECT(MammalObj)
         ASK METHOD Swim;
END OBJECT;

MermaidObj = OBJECT(HumanObj, FishObj)
END OBJECT;

• Here, MermaidObj has inherited all the fields as well as all the methods of both HumanObj and FishObj.
RESOLVING CONFLICTING FIELD NAMES

- If each of the parent objects have a field with the same identifier, then the child object will contain a field for each one.

- To reference such a field, assign a reference variable of the child type to a reference of one of the parent types.

example:

```plaintext
Man : HumanObj; (* has a field 'age' *)
Fish : FishObj; (* has a field 'age' *)
Mermaid : MermaidObj; (* derived from both above *)

... FishPart := Mermaid; (* declared as FishObj *)
HumanPart := Mermaid; (* declared as HumanObj *)
OUTPUT("age: ", FishPart.age; (* Fish part age*)
OUTPUT("age: ", HumanPart.age; (* Human part age *)
(OUTPUT("age: ", Mermaid.age; *)) (* run time error*)
```

- Hint: Try to avoid conflicting field names!!
RESOLVING CONFLICTING METHOD NAMES

- Methods with conflicting names must be overridden in the child type (and explicitly declared as OVERRIDE). The implementation part of the object type must specify the code for the replacement method.

- The implementation can either completely ignore the parent's underlying methods, or carry them over in form of a qualified inheritance call.

- OVERRIDE'ing methods is not restricted to this situation. Any inherited method can be changed or completely replaced in the implementation part of the object.
OVERRIDING METHODS

example:

```mermaid
TYPE
    MermaidObj = OBJECT(HumanObj, FishObj)
    OVERRIDE
        ASK METHOD Talk;
    END METHOD;
END OBJECT;

OBJECT MermaidObj;
    ASK METHOD Talk;
    BEGIN
        INHERITED FROM HumanObj Talk;
        INHERITED FROM FishObj Talk;
    END METHOD;
END OBJECT;
```
POLYMORPHISM

• Different objects can respond to the same message:

    ASK Human TO Talk;
    ASK Fish TO Talk;

• The objects can respond (behave) in different ways. All the 'outside world' has to know is the existence of the method and its parameters.

• This produces cleaner code and supports 'pluggability': you simply 'plug in' a different implementation of an object without having to change the entire program.

• This feature is supported by OVERRIDE'ing method implementations.
**DATA HIDING**

- Data can be protected and made invisible to other parts of the program.

- Recall: other object instances only have 'read' access to an object's fields.

- Fields and Methods can be declared as `PRIVATE`, thus even removing the 'read' access. Only the object instance itself can reference those fields and methods.

- In order to make the data available to other objects, a method has to be explicitly supplied to perform this function.
DATA HIDING (CONT.)

example:

```
TYPE
    TaxPayerObj = OBJECT
        ASK METHOD SetIncome(IN val: INTEGER);
        ASK METHOD ReportIncome: INTEGER;
    PRIVATE
        income: INTEGER;
    END OBJECT;

VAR
    TaxPayer : TaxPayerObj;
    x : INTEGER;
BEGIN
    NEW(TaxPayer);
    x := ASK TaxPayer TO ReportIncome;
    (* x := TaxPayer.income *) (* run time error!! *)
    ...
```
2. OBJECT - ORIENTED PROGRAMMING

2.3 Advanced Object - Oriented Features
ANYOBJ

- **ANYOBJ** is a special type to escape from strict type checking.

- **Note:** **ANYOBJ** should only be used where it is necessary to circumvent type checking!!

**Example:**

```pascal
PROCEDURE SwapObj(INOUT first, second: ANYOBJ);
VAR
    temp : ANYOBJ;
BEGIN
    temp := first;
    first := second;
    second := temp;
END PROCEDURE;
```

- **BEWARE!!** This procedure switches objects which must be of the same type! It does not switch objects of different types. But at least we can use it for all different object types.
OBJINIT & OBJTERMINATE (CONT.)

example:

```plaintext
TYPE
    BabyObj = OBJECT
        ASK METHOD ObjInit;
        ASK METHOD ObjTerminate;
    END OBJECT

VAR
    Baby : BabyObj;

OBJECT BabyObj;
    ASK METHOD ObjInit;
    BEGIN
        OUTPUT("Hello world");
    END METHOD;
    ASK METHOD ObjTerminate;
    BEGIN
        OUTPUT("Byebye world");
    END METHOD;
END OBJECT;
BEGIN
    NEW(Baby);
    DISPOSE(Baby);
END MODULE.
```
OVERRIDING OBJInit & OBJTerminate in Multiple Inheritance

- In multiple inheritance, it is necessary to override any existing ObjInit or ObjTerminate methods and invoke each of the parents ObjInit / ObjTerminate.

element:

```plaintext
BabyObj = OBJECT(ParentAObj, ParentBObj)
  ...
  OVERRIDE
  ASK METHOD ObjInit;
END OBJECT;
OBJECT BabyObj;
  ...
  ASK METHOD ObjInit;
BEGIN
  INHERITED FROM ParentAObj ObjInit;
  INHERITED FROM ParentBObj ObjInit;
END METHOD;
END OBJECT;
```
2. OBJECT - ORIENTED PROGRAMMING

2.4 Grouping Objects
MODSIM II GROUPS

- **QueueObj**  Provides a queue with FIFO order.
- **StackObj**  Provides a linked-list with LIFO order.
- **RankedObj** Provides a linked-list sorted by a user-defined criterion.
- **BTreeObj**  Provides a btree data structure (note: btree ≠ binary tree).

- These can all be imported from the library GrpMod.

- All Group objects are derived from BasicGroupObj defined in GrpMod.

- Basic operations include adding, searching and removing from a group.
ADDING OBJECTS TO A QUEUE

- A queue in MODSIM II is a linked-list where new elements are added at the back, thus First-In-First-Out.

element:

```
FROM GrpMod IMPORT QueueObj;
VAR
    Mercedes, Peugeot, Ferrari : CarObj;
    Ferry : QueueObj;
BEGIN
    NEW(Ferry);
    NEW(Mercedes); NEW(Peugeot); NEW(Ferrari);
    ASK Ferry TO Add(Mercedes);
    ASK Ferry TO AddBefore(Mercedes, Ferrari);
    ASK Ferry TO AddAfter(Mercedes, Peugeot);
END MODULE.
```
CONTENTS OF A QUEUE

example:

BEGIN
    ...
    IF Ferry.numberIn > 0
        DummyCar := ASK Ferry First();
        WHILE DummyCar <> NILOBJ
            OUTPUT(DummyCar.name);
            DummyCar := ASK Ferry Next(DummyCar);
        END WHILE;
    END IF;
    ...
END MODULE.

• Note: Next(DummyCar) does not remove the objects from the queue. It simply provides a reference to the next object.

• This example holds for any type of group object!
CONTENTS OF A QUEUE (CONT.)

- The **FOREACH** loop allows to browse an entire queue (and all other group objects).

- It takes as parameters a type identifier of the objects in the queue and the identifier of the queue itself. If it is empty, the statements enclosed by the loop will not be executed.

- Use the keyword **EXIT** to leave the loop.

example:

```plaintext
BEGIN
   ...
   FOREACH DummyCar IN Ferry
      OUTPUT(DummyCar.name);
   END FOREACH;
   ...
END MODULE;
```
REMOVING OBJECTS FROM A QUEUE

example:

```
BEGIN
  ...  
  WHILE Ferry.numberIn > 0
  DummyCar:= ASK Ferry TO Remove();
  DISPOSE(DummyCar);
  END WHILE;
  ...
END MODULE.
```

- **Remove()** removes the first element of a queue.
- The method **RemoveThis(IN member: #ANYOBJ)** can be used to remove a particular member of the queue.
- These methods also exits for **StackObj, RankedObj** and **BTreeObj**!
STACKS

- Stacks are linked-lists where new elements are added at the front. The last element to be added thus becomes the first one to be removed again (LIFO).

- StackObj is almost identical to QueueObj. It only has its Add() method overridden.

- Traversing through the contents of a stack or removing elements is implemented as shown with QueueObj.
RANKED GROUPS

- RankedObj allows the user to determine the order in which the elements of the group are sorted.

- When an object is added to a ranked group, its position is determined by the method `Rank(IN a, b: #ANYOBJ): INTEGER;` This method *must* be overridden!

- The method `Add(IN NewMember: #ANYOBJ);` automatically calls the method `Rank( . . )` to find the insertion position.
**Adding Objects to a Ranked Group**

- The new member is passed as the first argument to the `Rank( . . )` method.

- The second argument is always an existing member of the ranked group. The comparison sequence is as follows: compare the new element with the
  - first element
  - last element
  - 2nd, 3rd, 4th...

  in the ranked group.

- The ranked group is scanned until the insertion position is found. A rank result of -1 indicates insertion *before* the last compared element, a rank result of +1 or 0 indicates insertion *after* the last compared element.
**ADDING OBJECTS TO A RANKED GROUP (CONT.)**

example:

```plaintext
TYPE
    PersonObj = OBJECT
        name : STRING;
        ASK METHOD AssignName(IN nam: STRING);
    END OBJECT;

MyRankObj = OBJECT(RankedObj[ANYOBJ:PersonObj])
    OVERRIDE
    ASK METHOD Rank(IN a,b: PersonObj): INTEGER;
    END OBJECT;
```
example (cont.):

... (* PersonObj implementation in here *)
OBJECT MyRankObj;

(* sorts the elements by their names in descending order *)
ASK METHOD Rank(IN a,b: PersonObj): INTEGER;
BEGIN
   IF a.name < b.name
      RETURN(1);
   ELSIF a.name = b.name
      RETURN(0);
   ELSE
      RETURN(-1);
   END IF;
END METHOD;
END OBJECT;
**ADDING OBJECTS TO A RANKED GROUP (CONT.)**

**example (cont.):**

`VAR
  Person1, Person2, DummyPerson : PersonObj;
  RankGroup : MyRankObj;
  ...

BEGIN
  ...
  ASK Person1 TO AssignName("Peter");
  ASK Person2 TO AssignName("Janine");

  ASK RankGroup TO Add(Person1);
  ASK RankGroup TO Add(Person2);
  ...

  FOREACH DummyPerson IN RankGroup
    OUTPUT(DummyPerson.name);
  END FOREACH;

END MODULE.`
MORE ON RANKED GROUPS

- Removing objects from a ranked group and traversing it is similar to the corresponding operations for queues.

- `Remove()` removes the first member of the group.

- `RemoveThis(IN member: #ANYOBJ);` removes a particular member of the group.
MORE ON MODSIM II GROUPS

- All Group Objects in Modsim II are derived from BasicGroupObj and thus all own the following methods in addition to the ones discussed above:

ASK METHOD First();
ASK METHOD Last();
ASK METHOD Prev(IN candidate: #ANYOBJ): #ANYOBJ;
ASK METHOD Includes(IN candidate: #ANYOBJ): BOOLEAN;
**STATISTICAL GROUPS**

- The Objects StatQueueObj, StatStackObj, StatRankedObj and StatBTreeObj provide groups for which statistics are automatically collected.

- These have to be imported from GrpMod and behave like non-statistical group objects.

**Fields & Methods:**

```
ASK METHOD Count(): INTEGER;
ASK METHOD Maximum(): INTEGER;
ASK METHOD Mean(): REAL;
ASK METHOD Minimum(): INTEGER;
ASK METHOD Variance(): REAL;
ASK METHOD StdDev(): REAL;
ASK METHOD WtdMean(): REAL;
ASK METHOD WtdStdDev(): REAL;
ASK METHOD WtdVariance(): REAL;
```
2. OBJECT-ORIENTED PROGRAMMING

2.5 More Useful Libraries
INPUT / OUTPUT IOMOD

- Defines a StreamObj for streamed input / output.

Fields & Methods:

```plaintext
eof: BOOLEAN;
fileName: STRING;
ioResult: INTEGER;
ASK METHOD Close;
ASK METHOD Delete; (* do not use for an open stream *)
ASK METHOD Open(IN FileName: STRING; IN IOdirection: FileUseType);
ASK METHOD ReadChar(OUT ch: CHAR);
ASK METHOD ReadInt(OUT n: INTEGER);
ASK METHOD ReadLine(OUT str: STRING);
ASK METHOD ReadReal(OUT x: REAL);
ASK METHOD ReadString(OUT str: STRING);
ASK METHOD WriteChar(IN ch: CHAR);
ASK METHOD WriteInt(IN num: INTEGER; IN fieldwidth: INTEGER);
ASK METHOD WriteReal(IN num: REAL; IN fieldwidth, precision: INTEGER);
ASK METHOD WriteString(IN str: STRING);
```
INPUT / OUTPUT IOMod (cont.)

- Files have to be opened before executing the above methods. Possible options are Input, Output, InOut, Append, Update and CreateBinary - these are all FileUseType.

- "stdout", "stdin" and "stderr" are possible files to read from (usually the screen for standard output and keyboard for standard input).

- Writing to a non-existent file creates it.

- ReadString(...) reads a string up to the next white space.

- ReadLine(...) reads from the current position to the end of the line, excluding the newline.

- ReadChar(...) reads a single character and should not be used to read from the keyboard.
INPUT / OUTPUT IOMOD (CONT.)

- StreamObj owns further methods declared in the library IOMod for low-level handling of binary files:

  WriteBlock(IN buffer: ANYREC; IN size: INTEGER; IN blocknum: INTEGER);
  ReadBlock(IN buffer: ANYREC; IN size: INTEGER; IN blocknum: INTEGER);

  these can be used to create your own indexed binary files. (*Warning:* you have to manage the file structure such as block numbers yourself!)

- IOMod also defines a series of procedures useful for file handling:

  PROCEDURE DeleteFile(IN fname: STRING);
  PROCEDURE FileExists(IN fname: STRING): BOOLEAN;
  PROCEDURE FileSize(IN fname: STRING): INTEGER; (* in bytes *)
  PROCEDURE FileModTime(IN fname: STRING): INTEGER;(* time in secs*)
  PROCEDURE FileAccessTime(IN fname: STRING): INTEGER;(* time in secs*)
  PROCEDURE ReadKey(): CHAR; (* reads a single key, no CR required!*).
FILE MANAGER FILEMGR

- Defines the object FileObj to handle indexed file access.

Fields & Methods:

Name: STRING;
LastBlock: INTEGER;
BlockSize: INTEGER;
TableSize: INTEGER;
ASK METHOD Add(IN key: STRING; IN id: INTEGER; IN value: AnyRec;
IN size: INTEGER);
ASK METHOD Close;
ASK METHOD Create(IN name: STRING; IN blocksize: INTEGER);
ASK METHOD Find(IN key: STRING; IN id: INTEGER; IN value: ANYREC;
IN size: INTEGER): BOOLEAN;
ASK METHOD Next(IN value: ANYREC; IN size: INTEGER): BOOLEAN;
ASK METHOD FindId(IN id: INTEGER; IN value: ANYREC;
IN size: INTEGER): BOOLEAN;
ASK METHOD NextId(IN value: ANYREC; IN size: INTEGER): BOOLEAN;
ASK METHOD Update(IN size: INTEGER);
FILE MANAGER FILEMGR (CONT.)

- FileObj saves records in binary format to a file. The file itself is organised as a hash table. The size of the hash table can be set by SetTableSize(..).

- Before a record can be written to a file, the file has to be initialised using the method Initialize(IN name: STRING);

- Create(..) allows you to specify the size of the largest block to be stored.

- When a record is added to the file its position is determined by the method Hash(..).
FILE MANAGER FILEMGR (cont.)

- \texttt{Find(...) and Next(...) look for a record with identical hash key and Id.}

- The methods \texttt{FindId(...)} and \texttt{NextId(...)} are similar except that they search for the Id number of a record only.

- Note: the parameter 'value' for the methods find, next (and the like) points to a memory allocation to which the find results are copied if successful. Thus you have to \texttt{NEW} the identifier before it is passed to the methods as a function.
FILE MANAGER FILEMGR (CONT.)

- FileObj owns two methods for direct access to the blocks of the file:

  ASK METHOD ReadRec(IN value: ANYREC; IN size: INTEGER;
                        IN blocknum: INTEGER);

  ASK METHOD WriteRec(IN value: AnyRec; IN size: INTEGER;
                        IN blocknum: INTEGER);

  Writing a record to a file using this method does not find a place in the
  hash table! You have to keep track of the file management yourself, in
  particular block numbers (this is a warning - use Add (...) and
  Find (...)!).

- Note: the block number can be obtained by inspecting the field
  LastBlock of FileObj.
3. **OO DISCRETE EVENT SIMULATION**

3.1 Different Views of Simulation

3.2 Further Methods: TELL & WAITFOR

3.3 More Useful Libraries: RandMod & SimMod

3.4 Advanced Simulation Features
3. OO DISCRETE EVENT SIMULATION

3.1 Different Views of Simulation
**EVENT BASED SIMULATION**

- Event based simulation takes a global view on the simulation system and requires the explicit manipulation of events.

- Events describe the state changes of a system and thereby define its dynamics.

- Each event is implemented as a single routine. No time passes inside these routines.
Activity Based Simulation

- Activities are descriptions of the state changes in the system which are executed when certain conditions apply (i.e. each activity has a condition that determines when it is executed).

- The simulation engine does not need to keep an event list. Each time step, all the activities are scanned for their conditions and executed if these are satisfied.

- Focus is on the interaction between the activities.

```
Start -> Time Scan -> Activity scan -> Any action? -> Time up? -> Stop
```

PROCESS BASED SIMULATION

- A process is a sequence of events that pertain to a particular entity. The process describes the 'life' of an entity.

- The emphasis is put on the entities / processes that interact to determine the dynamics of the system.

- Processes are implemented as single routines in which time can elapse. The simulation engine hence needs to be able to interrupt and re-start a process to remember the current state of the entities.
**PROCESS BASED SIMULATION (CONT.)**

- The simulation engine usually keeps two kinds of process lists: the future event list (hold events known to be executed some time in the future) and the current event list (events to be executed at present plus all those waiting for a condition to happen).

- Processes in MODSIM II are described by **TELL** and **WAITFOR** methods.

- Each object can be involved in several processes at the same time.
MODSIM II's Pending List

- The system maintains a pending list of object instances which have activities scheduled.

- Object instances are sorted by the time of their first scheduled activity.

- The activity lists themselves are sorted by time.
**THE TIMING PROCEDURE**

- **StartSimulation**
  - Any Object with an Activity in the Pending List
    - **NO**
      - Return
    - **YES**
      - Select Object Activity with Earliest Execution Time
      - Update Simulation Clock to Time of Object Activity
      - Determine TELL Method to Execute
      - Remove Object's Activity from the Activity List
      - Execute the Object's TELL Method
CONTROLLING THE SIMULATION

- The timing procedure is started by issuing the command `StartSimulation` imported from `SimMod`.

- Simulation time is automatically maintained. `SimTime()` gives the current simulation time (imported from `SimMod`).

- `SimTime` is dimensionless. It can take any convenient unit, but make sure that all time parameters are in the same dimension.

- The timing procedure finishes when either the pending list is empty or on execution of the command `StopSimulation` (defined in `SimMod`).
3. **OO Discrete Event Simulation**

3.2 Further Methods: **TELL & WAITFOR**
TELL METHODS

- Whenever simulation time has to elapse, a TELL method must be used.

- They are scheduled asynchronously. Whenever a call for a TELL method is encountered, the method is scheduled for execution in the pending list. Control returns immediately to the first statement after the call.

- They can only take IN parameters (since there is no guarantee that at the time of return, the appropriate data structures are still allocated).
IMPLEMENTING A TELL METHOD

- **TELL** methods are implemented in the Object implementation part of the program.

example:

```plaintext
OBJECT FerryObj;
  TELL METHOD LoadCars(IN c1, c2: CarObj);
  BEGIN
    OUTPUT("Now loading cars. Takes 5 min. each");
    WAIT DURATION 5.0
    Add(c1);
    OUTPUT("first: ", SimTime());
    END WAIT;
    WAIT DURATION 5.0
    Add(c2);
    OUTPUT("second: ", SimTime());
    END WAIT;
  END METHOD;
  ...
```
**INVOKING A TELL METHOD**

- A **TELL** method is invoked by issuing the following syntax:
  
  ```plaintext
  TELL <object> TO <methodname> (* IN <time> *);
  ```

- The method is scheduled in the pending list and control returns immediately to the next statement.

**example:**

```plaintext
VAR
  Ferry : FerryObj;
  Ferrari, Truck : CarObj;
BEGIN
  NEW(Ferry); NEW(Ferrari); NEW(Truck);
  TELL Ferry TO LoadCars(Ferrari, Truck);
  StartSimulation;
END MODULE.
```
WAITFOR METHOD

- Combines the features of ASK and TELL methods: it can elapse simulation time and take return values.

- WAITFOR methods can only be invoked using a WAIT FOR statement (which must be used in either a TELL or a WAITFOR method itself).

- The invoking method will not continue to execute until the WAITFOR method terminates (and hence parameters can be returned).

- WAITFOR methods are implemented like TELL methods.
WAITFOR METHOD (CONT.)

description:

FROM SimMod IMPORT StartSimulation, SimTime;
FROM GrpMod IMPORT QueueObj;
TYPE
   CarObj = OBJECT
      name : STRING;
      ASK METHOD AssignName(IN str: STRING);
      WAITFOR METHOD DriveOnBoard;
END OBJECT;

   FerryObj = OBJECT(QueueObj)
      TELL METHOD LoadCars(IN c1, c2: CarObj);
      TELL METHOD Cruise;
END OBJECT;

VAR
   Ferrari : CarObj;
   Truck : CarObj;
   Ferry : FerryObj;
WAITFOR METHOD (CONT.)

example (cont.):

OBJECT CarObj;
    ASK METHOD AssignName(IN str: STRING);
    BEGIN
        name := str;
    END METHOD;
    WAITFOR METHOD DriveOnBoard;
        OUTPUT(name, " starts driving at ", SimTime());
        WAIT DURATION 5.0
        OUTPUT(name, " ends driving at ", SimTime());
    END WAIT;
    END METHOD;
END OBJECT;
WAITFOR METHOD (CONT.)

example (cont.):

```plaintext
OBJECT FerryObj;
  TELL METHOD LoadCars(IN c1, c2: CarObj);
  BEGIN
    OUTPUT("Now loading cars. Takes 5 min. each");
    WAIT FOR c1 TO DriveOnBoard;
      Add(c1);
    END WAIT;
    WAIT FOR c2 TO DriveOnBoard;
      Add(c2);
    END WAIT;
    Cruise;
  END METHOD;
  TELL METHOD Cruise;
  BEGIN
    OUTPUT("Now start cruising at ", SimTime());
    WAIT DURATION 20.0
      OUTPUT("Now have arrived at ", SimTime());
    END WAIT;
  END METHOD;
END OBJECT;
```
WAITFOR METHOD (CONT.)

example (cont.):

BEGIN
    NEW(Ferrari);
    NEW(Truck);
    NEW(Ferry);
    ASK Ferrari TO AssignName("Ferrari");
    ASK Truck TO AssignName("Truck");
    TELL Ferry TO LoadCars(Ferrari, Truck);
    StartSimulation;
END MODULE.
WAIT STATEMENT

- **WAIT** statement is used to suspend the execution whenever time has to elapse.

- The **WAIT** statement specifies a reason for wait, a sequence of statements to be executed upon successful completion and an optional sequence of statements to be executed if the wait is interrupted.

example:

```
WAIT reason
...
ON INTERRUPT
...
END WAIT;
```

- The reason can take 3 different forms:
  - **WAIT DURATION** time;
  - **WAIT FOR** SomeObj TO SomeMethod;
  - **WAIT FOR** SomeTrigger TO Fire;
WAIT DURATION

example:

```plaintext
WAIT DURATION 30.0
...
END WAIT;
```

- Most basic form of the `WAIT` statement.
- The specified time has to be of type `REAL`.
WAIT FOR SOMEOBJ TO SOMEMETHOD

example:

    WAIT FOR AirCraftObj TO Fly
    ....
    END WAIT;

• Suspends the execution of the current method until the method 'Fly' has been executed.

• Schedules the method 'Fly' in the pending list for execution.

• This is the only legitimate way to call a WAITFOR method.

• Useful for synchronising activities between objects.
WAIT FOR TRIGGER TO FIRE

- Triggers are signals that can be sent if a particular event happens.
- Implemented in form of TriggerObj.
- `WAIT FOR` suspends the execution of the calling method until the condition that fires the trigger is met, i.e. until the trigger is fired.
- This is an alternative means to synchronising the activities between objects.

example:

```plaintext
WAIT FOR Gun TO Fire
    ...
END WAIT;
```
TRIGGER OBJECTS (CONT.)

- TriggerObj is defined in the library module SimMod.

Methods:

ASK METHOD AddToList(IN a: ANYREC);
ASK METHOD Dump;
WAITFOR METHOD Fire;
ASK METHOD InterruptTrigger;
ASK METHOD NumWaiting(): INTEGER;
TELL METHOD Trigger;
ASK METHOD Release;
TRIGGER OBJECTS (CONT.)

example:

FROM SimMod IMPORT StartSimulation, SimTime, TriggerObj;
TYPE
  RefereeObj   = OBJECT
    TELL METHOD StartRace(IN shot: TriggerObj);
END OBJECT;
  SprinterObj = OBJECT
    Speed : REAL;
    Name  : STRING;
    ASK METHOD AssignName(IN str: STRING);
    ASK METHOD AssignSpeed(IN real: REAL);
    TELL METHOD Race(IN shot: TriggerObj);
END OBJECT;
VAR
  Referee   : RefereeObj;
  Sprinter1 : SprinterObj;
  Sprinter2 : SprinterObj;
  shot      : TriggerObj;
TRIGGER OBJECTS (CONT.)

example (cont.):

```
OBJECT RefereeObj;
    TELL METHOD StartRace(IN shot: TriggerObj);
    BEGIN
        OUTPUT("The olympic race starts in 5 seconds");
        WAIT DURATION 5.0
        ASK shot TO Release;
        END WAIT;
    END METHOD;
END OBJECT;
```
**Trigger Objects (cont.)**

example (cont.):

```plaintext
OBJECT SprinterObj;
    ASK METHOD AssignName(IN str: STRING);
        ... (* implementation comes here *)
    ASK METHOD AssignSpeed(IN real: REAL);
        ... (* implementation comes here *)
    TELL METHOD Race(IN shot: TriggerObj);
BEGIN
    WAIT FOR shot TO Fire
        OUTPUT(Name," starts to run at ", SimTime());
    WAIT DURATION (100.0/Speed)
        OUTPUT(Name," arrives at ", SimTime());
    END WAIT;
    END WAIT;
    END METHOD;
END OBJECT;
```
TRIGGER OBJECTS (CONT.)

example (cont.):

BEGIN
    NEW(Referee);
    NEW(Sprinter1);
    NEW(Sprinter2);
    NEW(shot);
    ASK Sprinter1 TO AssignName("Carl");
    ASK Sprinter2 TO AssignName("Ben");
    ASK Sprinter1 TO AssignSpeed(11.0); (* m/sec *)
    ASK Sprinter2 TO AssignSpeed(10.0);
    TELL Sprinter1 TO Race(shot);
    TELL Sprinter2 TO Race(shot);
    TELL Referee TO StartRace(shot);
    StartSimulation;
END MODULE.
INTERRUPTING OBJECT ACTIVITIES

• The full implementation of a **WAIT** statement specifies an **ON INTERRUPT** clause.

• **Interrupt** is a procedure defined in SimMod and it can take one of 4 forms:

  ```
  Interrupt(IN object: ANYOBJ; IN methName: STRING);
  InterruptAll(IN object: ANYOBJ);
  InterruptMethod(IN activity: ACTID);
  InterruptWaitingFor(IN activity: ANYREC);
  ```

• Interrupting a waiting activity will cause it to execute the **ON INTERRUPT** clause. If there is no such clause, a run time error results.

• Note: Triggers are interrupted using the **TriggerObj**'s method **InterruptTrigger**.
INTERRUPTING OBJECT ACTIVITIES (CONT.)

example:

FROM SimMod IMPORT StartSimulation, SimTime, TriggerObj, InterruptAll;

TYPE
  RefereeObj = OBJECT
    TELL Method StartRace(IN shot: TriggerObj);
  END OBJECT;

  SprinterObj = OBJECT
    Name : STRING;
    Speed : REAL;
    ASK METHOD AssignName(IN str: STRING);
    ASK METHOD AssignSpeed(IN real: REAL);
    TELL METHOD Race(IN shot: TriggerObj);
  END OBJECT;

  InterObj = OBJECT
    TELL METHOD Break(IN obj1, obj2: ANYOBJ;
                      IN time: REAL; IN shot: TriggerObj);
  END OBJECT;
example (cont.):

```plaintext
VAR
    Referee : RefereeObj;
    Sprinter1 : SprinterObj;
    Sprinter2 : SprinterObj;
    shot : TriggerObj;
    Virus : InterObj;
    VirTime : REAL;

OBJECT RefereeObj;
    TELL METHOD StartRace(IN shot: TriggerObj);
    BEGIN
        TELL shot TO Trigger IN 5.0;
        END METHOD;
    END OBJECT;
```
example (cont.):

```
OBJECT SprinterObj;
    ASK METHOD AssignName(IN str: STRING);
        ... (* implementation in here *)
    ASK METHOD AssignSpeed(IN real: REAL);
        ... (* implementation in here *)
    TELL METHOD Race(IN shot: TriggerObj);
BEGIN
    WAIT FOR shot TO Fire
        OUTPUT(Name," started run at ", SimTime());
        WAIT DURATION (100.0/Speed)
        OUTPUT(Name," arrived at ", SimTime());
    ON INTERRUPT
        OUTPUT(Name,": Oh no, I'm injured");
    END WAIT;
    ON INTERRUPT
        OUTPUT(Name,": Sorry! I'm going home now");
    END WAIT;
    END METHOD;
END OBJECT;
```
interrupting object activities (cont.)

example (cont.):

OBJECT InterObj;
    TELL METHOD Break(IN obj1, obj2: ANYOBJ;
        IN time: REAL; IN shot: TriggerObj);
    BEGIN
        WAIT DURATION time
        If time < 5.0
            ASK shot TO InterruptTrigger;
        ELSE
            InterruptAll(obj1);
            InterruptAll(obj2);
        END IF;
        END WAIT;
    END METHOD;
END OBJECT;
INTERRUPTING OBJECT ACTIVITIES (CONT.)

element (cont.):

BEGIN
    NEW(Referee);
    NEW(Sprinter1);
    NEW(Sprinter2);
    NEW(shot);
    NEW(Virus);
    OUTPUT("Please enter a time for the interrupt:");
    INPUT(VirTime);
    ASK Sprinter1 TO AssignName("Carl");
    ASK Sprinter2 TO AssignName("Ben");
    ASK Sprinter1 TO AssignSpeed(11.0);
    ASK Sprinter2 TO AssignSpeed(10.0);
    TELL Virus TO Break(Sprinter1, Sprinter2, VirTime,
    shot);
    TELL Referee TO StartRace(shot);
    TELL Sprinter1 TO Race(shot);
    TELL Sprinter2 TO Race(shot);
    StartSimulation;
END MODULE.
TERMINATE

- **TERMINATE** can be used from within a `WAIT FOR` statement to stop its execution.

- It terminates the execution of the current `TELL / WAITFOR` method, as well as the method from which it was called, as well as the method from which it was called etc..

- **Interrupt** is used from outside the objects method, **TERMINATE** is used from inside the objects' method.
3. OO DISCRETE EVENT SIMULATION

3.3 More Useful Libraries: RandMod & SimMod
**RANDOM MODULE RANDMOD**

- `RandomObj` provides methods to generate random numbers with different distributions. It is defined in `RandMod`.

**Fields & Methods:**

```plaintext
antithetic: BOOLEAN;
originalSeed: INTEGER;
currentSeed: INTEGER;
ASK METHOD Sample(): REAL;
ASK METHOD SetSeed(IN NewSeed: INTEGER);
ASK METHOD SetAntithetic(IN onOff: BOOLEAN);
ASK METHOD Reset();
ASK METHOD UniformReal(IN lo, high: REAL): REAL;
ASK METHOD UniformInt(IN lo, high: INTEGER): INTEGER;
ASK METHOD Exponential(IN mean: REAL): REAL;
ASK METHOD Normal(IN mean, sigma: REAL): REAL;
ASK METHOD Gamma(IN mean, alpha: REAL): REAL;
ASK METHOD Beta(IN alpha1, alpha2: REAL): REAL;
ASK METHOD Triangular(IN min, mean, max: REAL): REAL;
```
RANDOM MODULE RandMod (cont.)

Fields & Methods (cont.):

ASK METHOD SetAntithetic(IN onOff: BOOLEAN);
ASK METHOD Weibull(IN shape, scale: REAL): REAL;
ASK METHOD Erlang(IN mean: REAL; IN K: INTEGER): REAL;
ASK METHOD LogNormal(IN SDev, Mean: REAL): REAL;
ASK METHOD Poisson(IN mu: REAL): INTEGER;
ASK METHOD Binomial(IN n: INTEGER; IN p: REAL): INTEGER;

- RandMod provides furthermore two procedures: Random and FetchSeed.

- Random() returns a sample between 0.0 and 1.0 using the C-compilers random number generator.

- FetchSeed(IN SimscriptSeedNumber: INTEGER): INTEGER; returns the first seed of the given random number stream. These are described in detail below.
RANDOM NUMBER GENERATOR

- MODSIM II uses a Prime Modulus Multiplicative Linear Congruential pseudo random number Generator (PMMLCG) with period $2^{31}-2$ ($=2.147.483.646$).

- General formula:

  $$Z_i = (a * Z_{i-1}) \pmod{m}$$

  where $m=2^{31}-1$, $a=630.360.016$ and $Z_0=2.116.429.302$.

- With these coefficients, a period of $(m-1)$ is realised and each integer in the stream can be obtained exactly once.
RANDOM NUMBER GENERATOR (CONT.)

- MODSIM II provides 10 different seeds of length 100,000 each.

- Each RandomObj has a field that keeps its original seed and a field for the current seed. When a random number is drawn the current seed \( Z_i \) is advanced.

- The default seed is the Simscript seed number 1 \((=2,116,429,302)\), set by ObjInit.

- `FetchSeed(IN SimscriptSeedNumber: INTEGER): INTEGER;` allows to return the Simscript seeds (1-10), each 100,000 random numbers apart. This procedure is declared in RandMod.

- `SetSeed(IN NewSeed: INTEGER);` allows to set the original seed for a stream. This method is owned by RandomObj.
The field **antithetic** allows you to use the Antithetic Variates (AV) variance reduction technique.

AV tries to introduce negative correlation between pairs of runs in a simulation. If the random number $U_i$ is generated in one run, the value $(1-U_i)$ will be used in the next run.

Note: in order to really reduce the variance between successive runs, the random numbers have to be used for the same purpose (e.g. number of arrivals), so the runs must be synchronised.
MORE ON THE PENDING LIST

- The Pending List is implemented in two different forms: a linked-list and a calendar queue. The type can be determined by the procedure `UseCalendar(IN flag: BOOLEAN);` defined in `SimMod`.

- The activities that are included in the Pending List include:
  - methods scheduled for execution (through a `TELL` call)
  - methods in a `WAIT DURATION` state
  - methods which have been waiting for another method to finish and are now re-scheduled.

- Methods in a `WAIT .. FOR` state are kept in a separate list with no particular name. The number of those methods can be obtained by the procedure `NumWAITFOR` declared in `SimMod`. 
MORE ON THE PENDING LIST (CONT.)

- The linked-list structure uses a modified linear search for traversing the list.

- The calendar queue is implemented in form of a dynamic hash table where the contents of the table are the object instances.

- The hash function is again depending on the first execution time of the object instance's activity list.

- The default data structure for the Pending List is the calendar queue (simply because it's a lot faster!!)
SIMULATION CONTROL

- The library module SimMod further declares an object SimControlObj with fields and methods to influence the execution order of activities.

- If an instance of this object is created and the method SetTieBreaking is set to TRUE, the user can override the ChooseNext method.

- Similarly, if SetTimeAdvance is set to TRUE, the method TimeAdvance can be overridden to exhibit user-defined behavior.

Methods of SimControlObj:

ASK METHOD TimeAdvance(IN newTime: REAL): REAL;
ASK METHOD ChooseNext(IN group: ActivityGroup): ACTID;
ASK METHOD SetTieBreaking(IN flag: BOOLEAN);
ASK METHOD SetTimeAdvance(IN flag: BOOLEAN);
**SIMULATION MODULE SIMMOD**

- Defines Procedures and Objects to manipulate the Pending List.

Timescale : REAL;  { number of real seconds per simulation unit }
PROCEDURE StartSimulation;
PROCEDURE StopSimulation;
PROCEDURE NumActivities(IN object : ANYOBJ) : INTEGER;
PROCEDURE NumObjPending() : INTEGER;
PROCEDURE NumActPending() : INTEGER;
PROCEDURE NumWAITFOR() : INTEGER;
PROCEDURE ResetSimTime(IN newtime : REAL);
PROCEDURE ActivityOwner(IN activity : ACTID) : ANYOBJ;
PROCEDURE ScheduledTime(IN activity : ACTID) : REAL;
PROCEDURE ActivityName(IN activity: ACTID) : STRING;
PROCEDURE InterruptMethod(IN activity : ACTID);
PROCEDURE Interrupt(IN object : ANYOBJ; INmethName : STRING);
PROCEDURE InterruptWaitingFor(IN activity : ANYREC);
PROCEDURE InterruptAll(IN object : ANYOBJ);
PROCEDURE PendingListDump(IN DoActList: BOOLEAN);
PROCEDURE ActivityListDump(IN ProcObj: ANYOBJ);
PROCEDURE SimTime() : REAL;
PROCEDURE UseCalendar(IN flag : BOOLEAN);
3. **OO DISCRETE EVENT SIMULATION**

3.4 Advanced Simulation Features
**RESOURCE OBJECTS**

- **ResourceObj** provides a pool of elements to handle the modelling of resources. It is declared in ResMod.

- Resources may be acquired for a specific time period, they can be obtained with priority, their number can be increased, decreased etc..

- Resources must be created before they can be used.

- **ResourceObj** keeps a pending list for requests and an allocation list for all allocated resources of its pool.
RESOURCE OBJECTS (CONT.)

Fields & Methods:

MaxResources: INTEGER;
Resources: INTEGER;
PendingResources: INTEGER;
ASK METHOD Create(IN number: INTEGER);
ASK METHOD IncrementResourcesBy(IN incBy: INTEGER);
TELL METHOD DecrementResourcesBy(IN decBy: INTEGER);
WAITFOR METHOD Give(IN me: ANYOBJ; IN numberDesired: INTEGER);
WAITFOR METHOD TimedGive(IN me: ANYOBJ; IN numberDesired: INTEGER;
    IN timePeriod: REAL);
WAITFOR METHOD PriorityGive(IN me: ANYOBJ; IN numberDesired:
    IN priority: REAL);
ASK METHOD TakeBack(IN FromMe: ANYOBJ; IN numberReturned: INTEGER);
ASK METHOD Transfer(IN From, To: ANYOBJ; IN numberTrans: INTEGER);
ASK METHOD Cancel(IN object: ANYOBJ; IN numberToCancel: INTEGER);
RESOURCE OBJECTS (CONT.)

example:

FROM SimMod IMPORT StartSimulation, SimTime;
FROM ResMod IMPORT ResourceObj;

TYPE
  MechanicObj = OBJECT
    Name : STRING;
    ASK METHOD AssignName(IN str: STRING);
    TELL METHOD Repair(IN hammer: ResourceObj);
  END OBJECT;

VAR
  mechanic1 : MechanicObj;
  mechanic2 : MechanicObj;
  mechanic3 : MechanicObj;
  hammer : ResourceObj;
RESOURCE OBJECTS (CONT.)

example (cont.):

```object
OBJECT MechanicObj;
    ASK METHOD AssignName(IN str: STRING);
        ...
        (* implementation in here *)
    TELL METHOD Repair(IN hammer: ResourceObj);
    BEGIN
        OUTPUT(Name," waiting for hammer at ", SimTime());
        WAIT FOR hammer TO Give(SELF,1);
        OUTPUT(Name," starts repair at ", SimTime());
        WAIT DURATION 5.0
        OUTPUT(Name," finishes work at ", SimTime());
        OUTPUT(Name," gives hammer back");
        ASK hammer TO TakeBack(SELF,1);
        END WAIT;
    END METHOD;
END OBJECT;
```
RESOURCE OBJECT (cont.)

dexample (cont.):

BEGIN
  NEW(mechanic1);
  NEW(mechanic2);
  NEW(mechanic3);
  NEW(hammer);
  ASK hammer TO Create(1);
  ASK mechanic1 TO AssignName("Sylvester");
  ASK mechanic2 TO AssignName("Arnold");
  ASK mechanic3 TO AssignName("Bill");
  TELL mechanic1 TO Repair(hammer);
  TELL mechanic2 TO Repair(hammer);
  TELL mechanic3 TO Repair(hammer);
  StartSimulation;
  ... (* Dispose objects here *)
END MODULE.
RESOURCE STATISTICS

- Each ResourceObj can automatically maintain statistics on allocated and pending resources.
- Both types of statistics can be weighted with respect to simulation time.
- Invoking the ASK METHOD SetAllocationStats(IN on: BOOLEAN); with the parameter TRUE enables the collection of statistics.
RESOURCE STATISTICS (CONT.)

Further ResourceObj Methods:

ASK METHOD Reset;
ASK METHOD ResetAllocationStats;
ASK METHOD SetAllocationStats(IN on: BOOLEAN);
ASK METHOD AllocMaximum(): INTEGER;
ASK METHOD AllocMinimum(): INTEGER;
ASK METHOD AllocCount(): INTEGER;
ASK METHOD AllocMean(): REAL;
ASK METHOD AllocVariance(): REAL;
ASK METHOD AllocStdDev(): REAL;
ASK METHOD AllocWtdMean(): REAL;
ASK METHOD AllocWtdVariance(): REAL;
ASK METHOD AllocWtdStdDev(): REAL;

- For the equivalent pending statistics, replace 'Alloc' by 'Pending'
RESOURCE STATISTICS (CONT.)

dexample:

BEGIN
NEW(mechanic1);
... (* create instances of objects *)
ASK hammer TO Create(1); (* or 2, or 3 *)
ASK hammer TO SetAllocationStats(TRUE);
ASK hammer TO SetPendStats(TRUE);
ASK mechanic1 TO AssignName("Sylvester");
... (* initialise other mechanics *)
TELL mechanic1 TO Repair(hammer);
... (* Tell other mechanics to repair too *)
StartSimulation;
OUTPUT("Statistics for the hammer: ");
OUTPUT("Allocated Maximum: ", ASK hammer AllocMaximum);
OUTPUT("Allocated Minimum: ", ASK hammer AllocMinimum);
OUTPUT("Allocated Wtd Mean: ", ASK hammer AllocWtdMean);
... (* ask for other statistics *)
END MODULE.
MORE ON STATISTICS: MONITORING

- Monitor objects 'watch' other data types and automatically execute methods declared as monitored.

- There are 3 different types of executing monitor methods:
  
  left: when the data type is changed, i.e. upon 'write' access  
  right: when the data type is accesses, i.e. upon 'read' access  
  left & right: both of the above.

- Monitoring is mainly used for automatically collecting statistics. MODSIM II provides 4 basic monitoring objects: IStatObj, RStatObj, ITimedStatObj, RTimedStatObj, all of which are declared in StatMod.
DECLARING A MONITORED VARIABLE

example:

SomeReal : LMONITORED REAL BY RStatObj;
SomeInt : RMONITORED INTEGER BY IStatObj;

• Statistics on SomeReal and SomeInt will automatically be collected whenever these are accessed.
STATISTICAL MONITOR OBJECTS

Fields & Methods:

Count: INTEGER;
Maximum: INTEGER / REAL;
Minimum: INTEGER / REAL;
Sum: REAL;
SumOfSquares: REAL;
low: INTEGER;
high: INTEGER;
interval: INTEGER;
ASK METHOD SetHistogram(IN low, high, interval: INTEGER);
ASK METHOD GetHistogram(): histogram;
ASK METHOD Mean(): REAL;
ASK METHOD MeanSquare(): REAL;
ASK METHOD Variance(): REAL;
ASK METHOD StdDev(): REAL;
ASK METHOD Reset;
PRIVATE
Histogram: histogram;
REFERENCING STATISTICS OF A MONITORED VARIABLE

- The built-in procedure `GETMONITOR` provides a means to access the statistics of a monitored variable.

- `GETMONITOR(SomeReal, RStatObj)` returns the monitor object of the variable `SomeReal`.

- The relevant statistics are then obtained by referencing the fields / methods of the monitor object.

example:

```plaintext
max := GETMONITOR(SomeReal, RStatObj).Maximum;
mean := ASK GETMONITOR(SomeInt, IStatObj) Mean;
```
REFERENCING HISTOGRAMS

- The predefined type histogram is used by monitor objects to create histograms. It is declared in StatMod as an ARRAY INTEGER OF REAL.

- The fields low, high and interval specify the setup of the histogram.

- Since histogram is a private field, it can only be accessed by the method GetHistogram():

example:

```pascal
VAR
   MyHist : histogram;
   ...
BEGIN
   ...
   MyHist := ASK GETMONITOR(SomeReal, RStatObj) TO GetHistogram();
   ...
```
USER DEFINED MONITORING

• MODSIM II allows you to create your own monitoring objects. This might be useful for debugging or whenever you want to customize the behaviour of the monitor object.

• A user defined monitor object must be declared in the type declaration section of the program and must state what data type is being monitored.

example:

```
IntMon = MONITOR INTEGER OBJECT
         ... (* other fields and methods *)
         LMONITOR METHOD leftAccess;
         RMONITOR METHOD rightAccess;
         END OBJECT;
```
USER DEFINED MONITORING (cont.)

- The monitoring methods must be implemented in the object implementation section of the program.

- The function `VALUE` provides the last value of the monitored variable or field.

- The function `NEWVALUE` returns the value to which the monitored variable or field is going to be updated.

- The function `UPDATEVALUE(\text{argument})` changes the `NEWVALUE` to the argument given in `UPDATEVALUE`.

- `NEWVALUE` and `UPDATEVALUE` are only allowed in left monitor methods.

- To control monitoring use the procedures `ACTIVATE(\text{monitored variable, monitor object})` and `DEACTIVATE(\text{monitored variable, monitor object})`. 
USER DEFINED MONITORING (CONT.)

example:

OBJECT IntMon;
   LMONITOR METHOD leftAccess;
BEGIN
   OUTPUT("oldvalue = ", VALUE);
   OUTPUT("newvalue = ", NEWVALUE);
   IF (VALUE > 0) AND (NEWVALUE > VALUE*2)
      UPDATEVALUE(VALUE*2);
      OUTPUT("value updated to ", NEWVALUE);
   END IF;
END METHOD;

(* other methods *)

END OBJECT;
VAR
   someInt : LMONITORED INTEGER BY IntMon;
BEGIN
   someInt := someInt * 4;
   ...
END MODULE.