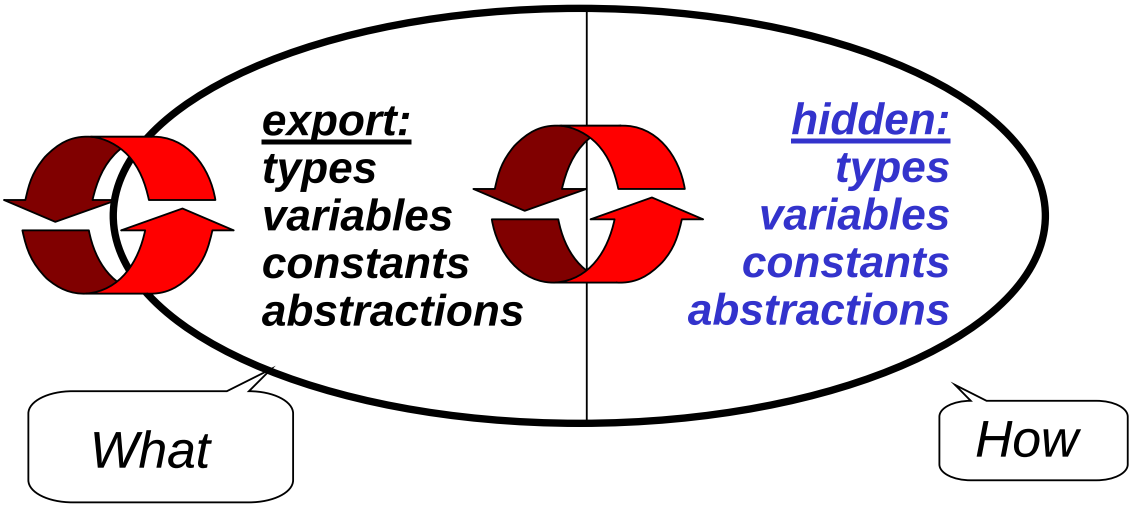
Encapsulation and Generics

Encapsulation

- **Programming in the large:** decomposition of a program into its components (or software units).
- **Module:** a software unit.
  - In some languages, the term *module* is used for a very specific abstraction mechanism, but we will stick to the more general meaning of the term.
- **Abstraction:** Each module should represent an abstraction, built using an abstraction mechanism, relying on some lingual metaphor.
  - **Separation of concerns:** ***what*** does the module do, vs. ***how*** does it do it.
- **Encapsulation:** mechanism for
  - **Defining module *boundaries***
  - **Information hiding:** maintaining the separation of concerns

Programming in the Large

- Each module encapsulates its components



Kinds of Modules

- Function/Procedure
- Package
- Abstract types
- Objects
- Classes
- ...

Simple Ada Packages

- **Simple packages:** everything is exported
- **Syntax: package** I **is** D **end** I**;**

```
package physics is
  c: constant Float := 3.0e+8;
  G: constant Float := 6.7e-11;
  h: constant Float := 6.6e-34;
end physics;
```

The `physics` package encapsulates a list of bindings.

At this stage, this is nothing more than a name space mechanism, very similar to the one available in C++.

The bindings can be used as `physics.c`, `physics.G` and `physics.h`

Packaging also Types and Variables...

```
package Earth is
  type Continent is
        (Africa, Antarctica, Asia, Australia,
         Europe, NorthAmerica, SouthAmerica);
  radius : constant Float := 6.4e6;
  area   : constant array (Continent) of Float :=
           (30.3e9, 13.0e9, 43.3e9, 7.7e9,
            10.4e9, 24.9e9, 17.8e9);
  population : array (Continent) of Integer;
end Earth;
```

Using the package:

```
for cont  in Earth.Continent loop
    put (Earth.population(cont) / Earth.area(cont));
end loop;
```

Packaging in ML

```
Structure I =
     struct
          D
     end
```


Information Hiding

- An important aspect of encapsulation: *preventing* the user of a module from accessing the module's internals.
  - **Functions and procedures:** block scope rules
  - **Packages**: body declarations.

```
package trig is
  function sin (x : Float) return Float;
  function cos (x : Float) return Float;
end trig;

package body trig is
     pi : constant Float := 3.1416;
     function norm (x : Float) return Float is
     ...;  -- return x modulo 2*pi
     function sin (x : Float) return Float;
     ...;  -- return the sine of norm(x)
     function cos (x : Float) return Float;
     ...;  -- return the cosine of norm(x)
end trig;
```

Information Hiding using Block Declarations

```
structure trig = struct
  local
     val pi = 3.1416;
       fun norm (x: real) =
            ... (* compute and return x modulo 2*pi *)
    in
       fun sin (x: real) =
            ... (* compute and return sine of norm(x) *)
       and cos (x: real) =
            ... (* compute and return cosine of norm(x) *)
       end
end
```

- **Using this ML structure:**
  `trig.cos(theta/2.0)`

Abstract Types

- An abstraction mechanism using the *primitive type* metaphor.
  - **Defined by their *properties*, rather than by a set of values.**
- **A case study:** *rational numbers*

  R = { rat(m,n) | m,n are integers }

  the definition is not good enough for practical purposes, e.g.

  rat(1,2) = rat(2,4) ?!

  An improved definition:

  R = { rat(m,n) | m,n are integers, n is positive and m and n have no common factor}

  This definition cannot be derived directly from basic types!

Why Abstract Types?

- Using the type metaphor is a powerful abstraction mechanism. Not all sophisticated types can be represented using the usual type creation operators.
- Problems in using a *representation type* for an *abstract* type (think: *list of elements* for a *set,* or *pairs of integers* for *rationals)*
  - **Extra values:** some values of the representation types are illegal for the abstract type.
  - **Duplicate values:** some distinct values of the representation type are identical as far as the abstract type is concerned.
  - **Mixing types:** values of the representation types could be mixed with other values of the representation type which are not intended to support the abstract type.
  - Solution**: use information hiding in an abstract type abstraction mechanism.**

Rational Numbers as Abstract Type (ML)

```
abstype rational = rat of (int * int)
with
  val zero = rat (0, 1)
  and one = rat (1, 1);

  fun op // (m: int, n: int) =
          if n <> 0
          then rat (m, n)
          else ... (* invalid rational number *)

   and op ++ (rat (m1,n1): rational,
              rat (m2,n2): rational) =
             rat (m1*n2 + m2*n1, n1*n2)

   and op == (rat (m1,n1): rational,
              rat (m2,n2): rational) =
             (m1*n2 = m2*n1)
end
```

The `rational` Abstract Type

- **Declared bindings (exported):**

`rational`	☞	an abstract type,
`zero`	☞	the rational number equal to 0,
`one`	☞	the rational number equal to 1,
`//`	☞	a `rational` ***constructor***,
`++`	☞	addition operation,
`==`	☞	equality test.

- **Use examples:**

```
val h = 1//2;
 if one ++ h == 6//4 then ... else ...
```

- **Deconstructor**

```
fun float (rat(m,n): rational) =  m / n
```


Objects

- **Object:** *a hidden variable with a set of exported operations*
- **Within a module, or a module by itself**
- **Supported directly by several languages (Ada, Smalltalk, C++, Java, Eiffel)**
- **Class gives a general declaration that can be instantiated to many objects (more later)**

Single Objects

- **Package declaration:** the export

```
package directory_object is
   procedure insert (newname    : in  Name;
                     newnumber  : in  Number);
   procedure lookup (oldname    : in  Name;
                     oldnumber  : out Number;
                     found      : out Boolean);
end directory_object;
```

The lifetime of the variable defined by this package is the same as that of the smallest block containing it.

Package Body: the Implementation

```
package body directory_object is
   type DirNode;
   type DirPtr is access DirNode;
   type DirNode is record
                    entryname   : Name;
                    entrynumber : Number;
                    left, right : DirPtr;
                  end record;
   root : DirPtr;
   procedure insert (newname    : in  Name;
                     newnumber  : in  Number) is
      ...;   -- add a new entry
   procedure lookup (oldname    : in  Name;
                     oldnumber  : out Number;
                     found      : out Boolean) is
      ...;   -- find an existing entry
begin
      ...;   -- initialize the directory
end directory_object;
```

Using the Object

- **Explicit use**

```
with directory_object;
directory_object.insert (me, 4955);
...
directory_object.lookup (me, mynumber, ok);
```

- **Implicit use**

```
with directory_object;
use  directory_object;
insert (me, 4955);
...
lookup (me, mynumber, ok);
```


Abstract Types *vs.* (Ada) Object Classes

- Common:
  - **Create several variables of similar structure**
  - **Representation is hidden**
  - **Access only by declared operations**
- Different:
  - **Notation**: abstract types support ordinary procedure call. Objects call for "sending a message".
  - **Abstract types can be used in the functional paradigm;** in contrast, objects are variables that can be updated!
    - Since objects cannot be returned from functions, it is impossible to declare a function that would take an immutable object and return a modified version of it.

Generics

- **Generic abstraction:** abstraction over a declaration
- **The generic body is elaborated (i.e. produces the bindings) in each generic instantiation**
- **Generics help increase reuse**
- **The instantiation is (usually) done at compile time, so the possible parameters to a generic abstraction are limited**
- **When there are no parameters, the Generic version (in Ada) is a class and the instantiation is an object**

Object Classes in Ada

- Object classes are created by making a generic package which takes ***no*** parameters.

```
generic package directory_class is
   ...
end directory_class;

package body directory_class is
      ...;
begin
      ...;
end directory_class;
```

Objects from Object Classes

- An Ada object is created by instantiating a generic package.
  - **package** homedir **is new** directory_class;
  - **package** workdir **is new** directory_class;
- Using objects:
  - workdir.insert(me, 4955);
  - homedir.insert(me,8715);
  - workdir.lookup(me,mynumber,ok)

Generics taking Value Parameters

- **The abstraction principle:** it should be possible to abstract over any syntactic class.
  - **Abstraction over a declaration:** a generic.
  - **Abstraction with parameters:** a generic taking a parameter.
- **Parameters:**
  - Usually must be known at compile time (it is difficult to generate declarations at runtime).
  - Simplest parameter: constant
  - Another kind of parameter: type

Generic Package Declaration (with Value Parameter)

```
generic
   capacity : in Positive;
package queue_class is
   procedure append (newitem : in  Character);
   procedure remove (olditem : out Character);
end queue_class;

package body queue_class is
   items: array (1..capacity) of Character;
   size, front, rear: Integer range 0..capacity;

   procedure append (newitem : in  Character) is
      ...; -- add newitem to the rear of the queue

   procedure remove (olditem : out Character) is
      ...; -- remove olditem from the front of the queue
begin
      ...; -- empty the queue
end queue_class;
```


Generic Instantiation

```
package line_buffer is new queue_class (120);
package terminal_buffer is new queue_class (80);
```

- **Elaboration order:**
  - **package parameter binding (**`capacity <--> 120 or 80`**)**
  - **produce the specific package (containing the appropriate array)**
  - **denote the package by its name (**`line_buffer or terminal_buffer`**)**

Generics with Type Parameters

- Applying the correspondence principle:
  - **It should be possible to have "type parameters".**
- It is difficult to (safely) implement run time type parameters.
  - **Therefore, we must have generics which take Type Parameters!**
- More generally, applying the abstraction principle:
  - **Given a piece of code:** Declaration, Routine, Class, …
  - **Make code applicable for *many types*.**
  - **The usual reuse benefits:**
    - Better overall quality
    - Lower maintenance cost
    - Save development efforts

Type Parameters

```
generic
   capacity : in Positive;
   type Item is private;
package queue_class is
   procedure append (newitem : in  Item);
   procedure remove (olditem : out Item);
end queue_class;

package body queue_class is
   items:  array (1..capacity)  of Item;
   size, front, rear: Integer range 0..capacity;

   procedure append (newitem : in  Item) is
      ...; items(rear) := newitem; ...;

   procedure remove (olditem : out Item) is
      ...; olditem := items(front); ...;
begin
      ...; -- empty the queue
end queue_class;
```

More on Type Parameters

- **Instantiation of the parameterized package:**

```
package line_buffer is
        new queue_class (120, Character);
    ...
    line_buffer.append('*');
```

- **A type parameter is less 'manageable' than value or variable parameters (whose type is known).**
- **At least, the assignment operator must be valid for the parametric type**
  - **this is achieved by the phrase**

```
type Item is private;
```


Type Parameters with Operations

- **A type declaration may contain a set of applicable operations for the type.**
- **The compiler has to check for each type parameter T:**
  - **for each generic instantiation:**
    - *every operation specified as applicable to T is also applicable to the argument type*
  - **for each generic abstraction:**
    - *every operation used for T in the generic abstraction is also specified as applicable to T*
- **(Formal) type parameters may be parameterized by themselves!**
  - In other words, there might be a relationship between the type parameters.
  - *See example on next foil…*

Type Parameters with Operations (Example)

A parameterized parameter!!!

```
generic
   type Item is private;
   type Sequence is
            array (Integer range <>) of Item;
   with function precedes (x, y : Item) return Boolean;
package sorting is
   procedure sort (seq : in out Sequence);
   procedure merge (seq1, seq2 : in Sequence;
                    seq : out Sequence);
package body sorting is
   procedure sort (seq : in out Sequence) is
   begin
      ...
      if precedes (seq(j), seq(i)) then ...
   end;
   procedure merge (seq1, seq2 : in Sequence;
                    seq : out Sequence) is
      ...
end sorting;
```


Elaborating Type Parameters

- **Case 1: using a primitive type and its operations**

```
type FloatSequence is
   array (Integer range <>)  of Float;

package ascending is
   new  sorting (Float, FloatSequence, “<=“);

package descending is
   new  sorting (Float, FloatSequence, “>=“);
```

Elaborating Type Parameters

- **Case 2: using a new type and its operations**

```
type Transaction is record ... end record;
type TransSequence is
   array (Integer range <>)  of Transaction;
function earlier (t1, t2 : Transaction)
                             return Boolean is
    ...;  -- return true iff t1 has an earlier time-tag than t2

package transaction_sorting is
    new sorting (Transaction,
                        TransSequence, earlier);
```


Static Typing and Genericity

- **Dynamically-typed languages:**
  - **Any variable/expression/operation can have different types**
  - **It is the programmer's responsibility to ensure that no run-time type error occurs**
- **Statically-typed languages: a** two player game
  - **Programmer:**
    - Specify type of participants
  - **Compiler:**
    - Check that the code is used only with participants of the right type
    - Translate the code to target language using this assumption.
      - **More efficient code**

    In dynamically typed languages, these two tasks are done in run time
- **Genericity:** Make the same piece of code usable for many different types, without compromising type safety and run time efficiency as in dynamic binding

Example: Function Template in C++

"A clever kind of macro that obeys the scope, naming, and type rules of C++." (Helpful oversimplification of B. Stroustrup)

- **Compactness and generality of macros**
- **Type safe**
- **Easy to write**

```
//  A template of a function to compute the maximum
//  of two elements of any type:
template <class AnyType>
  AnyType max(AnyType &a, AnyType &b)
  {
      return a > b ? a : b;
  }
```


Some Useful Function Templates

- Avoid redundant definition of **operator**`!=` when **operator**`==` is given:

```
template <class T>
   inline bool operator != (const T& x, const T& y) {
       return !(x == y);
   }
```

- Avoid redundant definitions of **operators** >, >= and <= when **operator**< is given:

```
template <class T>
   inline bool operator >(const T& x, const T& y) {
       return y < x;
   }
template <class T>
   inline bool operator <=(const T& x, const T& y) {
       return !(y < x);
   }
template <class T>
   inline bool operator >=(const T& x, const T& y) {
       return !(x < y);
   }
```

Taxonomy of Genericity

Genericity can be thought of as generating source by running routines at the compilation level.

- **Kind of routines:** Which lingual constructs can be parameterized over?
- **Arguments**: What kinds of parameters are allowed?
- **Type of arguments**: What constraints can be placed on parameters?
- **Call to routine**: How and when is the generic invoked?
- **Nested call**: Can the generic invoke other generics?
- **Environment of evaluation**: At which point is the generic instantiated?
- **Conditionals**: Can the body of a generic be different for different generics?
- **Name overloading**: Is it possible to re-use the name of a generic?

Example: Class Template in C++

```
template<typename Type>
 class Stack {
  Type buff[50];
  int sp;
 public:
  Stack(void): sp(0) {}
  void push(const Type &e) {
   buff[sp++] = e;
  }
  Type pop(void) { return buff[--sp]; }
  int empty(void) const { return sp == 0; }
  int full(void) const{ return sp == 50; }
 };
```

Class Template Member Functions

- Here's the syntax for the definition of member function of a template:

```
template <class Type>
   class Stack {
    int sp, size;
    Type *buff;
   public:
    Stack(s);
    void push(Type e);
   ...
};
```

```
template <class T>
   Stack<T>::push(T val){
     if (sp >= size) {
      assert(sp == size);
      error("Stack is full");
      return;
     }
     buff[sp++] = val;
   }
```

- Stack size is set in construction time
- Names of formal arguments to template are not necessarily consistent among definitions and declaration.

Using the Stack Template

```
void f()
{
 Stack<int> x;
 Stack<char *> y;

 y.push("life, universe, everything");
 x.push(42);

 ...

 if (!x.empty() && !y.empty()){
   char *question = y.pop();
   int answer = x.pop();
 }

 // etc.

}
```

Kinds of Parameters

Since generics are "routines executed at compile time and produce source code". Therefore, all parameters must be entities that are known at compile time.

- **Type**: most frequent and most important. All languages which support genericity allow type parameter.
- **Numerical Constant**: e.g., the integer constant 3.
  - **Useful for building generic arrays.**
- **Variables**:
  - **Constant: Address of a variable in C++**
- **Routine**:
  - **Constant: Address of function in C++**

In C++: both class templates and function templates can take any constant argument.

More?

- There are many variants on generics and templates
- As they get more and more complicated, they are less and less used…
- Still, they are useful and needed for generic classes
- Similar results can sometimes be obtained using abstract classes, inheritance, and subclasses (the kinds of polymorphism we have seen before)