Program modularization arose from the necessity of splitting large programs into fragments in order to compile them. . . . It was soon realized that modularization had great advantages in terms of large-grain program structuring.

Cardelli, 1996
Modularity

The maxims of modularity:

- minimize coupling — as independent as possible
- maximize cohesion — as focused as possible


A *module* is a construct for encapsulation, information hiding, and separate compilation, consisting of an interface specification and an implementation. It is a compile-time abstraction.

- encapsulation, aggregation – collecting pieces into a unit.
- information hiding – limiting access to the details
- representation independence – a change in the details does not affect the client
- separate compilation – capability to compile programs incrementally
In the shadow of many exciting development there has been a tendency to overlook the original purpose of modularization. Some language definitions specify what are to be the compilation units (e.g.: Ada), but others do not (e.g.: Standard ML). A paradoxical question then arises: when does a module system really support modularization (meant as separate compilation)?

Cardelli, 1996
Modularity

Each module should have well-defined *interface* or boundary. A *specification* is a description of the behavior of a module. Such as description may be informal as in English or formal as in ANNA, logic, etc.
In other languages such a construct is known as a *package* or a *structure*.
Ada, Modula 2, and Modula 3 interfaces are textually separate; Oberon’s are not. Ada and Modula 2 allow nested modules; Modula 3 and Oberon do not.
C/C++ header files

Object-oriented classes
Modules

Mesa, Clu the first?

*Modules provide a capability for partitioning a large system into manageable units. They can be used to encapsulate abstractions and to provide a degree of protection. In the design of Mesa, we were particularly influenced by the work of Parnas, who proposed information hiding as the appropriate criterion for modular decomposition, and by the concerns of Morris regarding protection in programming languages.*

Beschke, Morris Jr., and Satterthwaite,
CLU

complex = cluster is make_complex, real_part, imaginary_part, plus, times

rep = struct [ re, im : real ];
make_complex = proc (x, y: real) returns (cvt)
    return (rep$(re:x, im:y));
end make_complex;
real_part = proc (z: cvt) returns (real)
    return (x.re);
end real_part;
imaginary_part = ...
plus = proc (z, w: cvt) return (cvt)
    return (rep$(re:z.re+w.re, im:z.im+w.im));
end plus;
times = ...
end complex

cvt is reserved word calling for a conversion between the abstract type and its representation.
Modules

- interface specification – description of the items in the package
- implementation – programs, data structures, etc. to complete the facilities described in the interface
- client – some program that makes use of the facilities in the package

Modula 2, Modula 3, and Ada guarantee no violation of type security no matter the organization of the files or order of compilation.
C and C++ do not do inter module type checking!

link/main.c
link/aux.c

gcc -c aux.c # compile "aux.c", no errors

gcc aux.o main.c -o main # link "main.o" and "aux.o", no errors

main # run main, !!!!
0.000000 -9977185468416087696201563466699722509886527768824
with List; -- import other module
package Main is -- this module
begin
  null;
end Main;

In the Ada language identifiers are not case sensitive, so Main, mAiN, and main are the same. The Ada language does not specify how the computer retains modules, so, in general, an Ada implementation needs to know the correspondence between the name of the module and the operating system “handle” (the filename) of the module and its compiled artifacts.
Modules in Ada

GNAT requires the unit (module) name to be the same as the filename. It also uses a unit search path, cf., 3.3 Search Path of the GNAT user guide, as in the gcc.

2.3 File Naming Rules

The default file name is determined by the name of the unit that the file contains. The name is formed by taking the full expanded name of the unit and replacing the separating dots with hyphens and using lowercase for all letters.

N.B. lowercase!
Aggregation in Ada

Note separation of specification and implementation. Note lack of cohesion (this is just an example, do not program this way).

package/junk.ads
package/junk.adb
package/use_junk.adb
Aggregation in Java

- Packages: aggregation of classes
- Classes: aggregation of values, classes

No clear separation; classes also can be used to aggregate classes.

class Aggregate {
    class B { /*...*/ }
    class C { /*...*/ }
}
Type Representation

- Integer: twos complement, sign/magnitude; cf package/sign_integer_package.ads
- Real: IEEE 754, ...
- Stack: array, linked list
- Graph: 2D array, adjacency list

The user does not care about the implementation (information hiding).
Consider first the well-known concepts of procedure specification and procedure implementation. Here is an example of a procedure specification:

```c
// Push an element into the stack
void push (int i);
```

Here is an example of a procedure implementation (body):

```c
void push (int i) { data[size++] = i; }
```

What is a type specification? What is a type representation?
Often the only external interface to a type is its name.

```pascal
type T;  type T is (Red, Green, Blue);
```

```pascal
type T;  type T is array (1..3) of Positive;
```

```pascal
type T;  type T is record
  Top: Natural;
  Data: array (1..20) of Character;
end record;
```
Type Representation

But types in Ada can have (value) parameters. This is specified thusly:

```plaintext
type T (Size: Positive);
```

The implementation of the type might be:

```plaintext
type T is record
  Top: Natural;
  Data: array (1..Size) of Character;
end record;
```
Abstract Data Types

Modules usually focus, center around one type (cohesion).

Abstract data type. An abstract data type is a data type together with a complete set of relevant operations that form a new conceptual type of values. The representation of the type and the implementation of the operations are not important to the client and could be changed or improved.

Languages typically have two variations. Opaque data type. An opaque data type is one whose structure or representation is hidden from the client.

Transparent data type. A transparent data type is one whose structure or representation is visible to the client.
Abstract Data Types

Some operations have special purposes, so they have special names.

*Constructors*. Procedures or functions that construct (allocate memory for) values out of their subparts are called *constructors*.

*Predicates or recognizers*. Boolean functions to distinguish the different ways values are constructed are called *predicates*.

*Destructors or selectors*. Functions that take constructors apart and return their constituent parts are called *destructors*.

And, finally, and less common:

*Iterators*. Procedures that access all the elements individually of data structure like a list are called *iterators*. 
Specification/Implementation

with P

package P is
specification
end P;

package body P is
implementation
end P;

clients

*.ads

*adb

information hiding
Transparent Types

package P is
  type T is ...;
end P;

package body P is
end P;

clients know how T is implemented

transparent type
Opaque Types

package P is
  type T is private
 specification
  private
    type T is ... end P;

implementation

private
  type T is ... end P;

opaque type

*.ads

*.adb
Abstract Data Types

How to get an opaque type

- Ada: private section
- Modula-3: partial revelation
- SML: opaque type constraint
- Java, C++: declare members private
- Haskell: explicitly denying names to export
Haskell modules

In file Main.hs:

```haskell
module Main (main) where
import Tree (Tree(Leaf,Branch), fringe)

main = print (fringe (Branch (Leaf 1) (Leaf 2)))
```

In file Tree.hs:

```haskell
module Tree (Tree(Leaf,Branch), fringe) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)

fringe :: Tree a -> [a]
fringe (Leaf x) = [x]
fringe (Branch left right) = fringe left ++ fringe right
```
Haskell modules

```haskell
module Stack (  
    Stack, empty, isEmpty, push, top, pop) where

empty :: Stack a
isEmpty :: Stack a -> Bool
push :: a -> Stack a -> Stack a
top :: Stack a -> a
pop :: Stack a -> (a, Stack a)

newtype Stack a = StackImpl [a]  -- opaque!

empty = StackImpl []
isEmpty (StackImpl s) = null s
push x (StackImpl s) = StackImpl (x:s)
top (StackImpl s) = head s
pop (StackImpl (s:ss)) = (s,StackImpl ss)
```
Abstract Data Types

ada/programs/fraction/fraction_package.ads
ada/programs/fraction/fraction_package.adb
ada/programs/fraction/opaque.ads
ada/programs/fraction/opaque.adb
m3/programs/rational/src/Rational.i3
m3/programs/rational/src/Rational.m3
m3/programs/opaque/src/Rational.i3
m3/programs/opaque/src/Rational.m3
sml/programs/rn.sml
Representation Independence

The compiled client does not know what the actual representation of the data type.

- Ada: any type in private section
- SML: any type
- Modula-3: REF types only
- Modula-2: REF types only

Pointers all have same size (regardless of what they point to), so clients may be isolated from changes in representation if the type is a pointer type.
Abstract data type versus abstract state encapsulation. A package in Ada may be like an abstract data type and provide (export) a single type with relevant operations on that type. Or, a package in Ada may encapsulate state by holding variables.

fraction/rational.adt.ads
fraction/rational.adt.adb
fraction/rational_number_package.ads
fraction/rational_number_package.adb

A stack package could be done either way.
Modules versus Classes

compile-time abstraction       run-time
statically instantiated       dynamically
collection                   template
control visibility            extension
separate compilation

See Clemens Szyperski, "Import is Not Inheritance."
Dynamic loading in Java gives rise to the possibility of “no such method” error (despite the type checking).

module/Main.java
module/C.java
module/version/C.java
module/bad/C.java
The following steps correctly compile and then fool Java.

> rm *.class
> javac Main.java
> java Main
version 1
> cp version/C.class .
> java Main
version 2
> javac Main.java
> java Main
version 2
> cp bad/C.class .
> java Main
Exception in thread "main" java.lang.NoSuchMethodError: C.<init>(IC)V
    at Main.main(Main.java:5)
Object-Oriented Programming

A confusing collection of motivations.

▶ inheritance
▶ overriding
▶ dynamic dispatch
▶ narrowing versus widening
▶ implementation

Swiss-army knife approach: complex structure that does almost everything. Lego approach: simple pieces that fit together well.
I find OOP technically unsound... philosophically unsound... [and] methodologically wrong.

Alexander Stepanov, developer of the C++ STL
C++ was invented because Vogon poetry wasn't destructive enough.

Anonymous
Object-Oriented Programming

“Before the industrial revolution, the firearms industry was a loose coalition of individual craftsmen. Each firearm was crafted by an individual gunsmith. The revolution was sparked when Eli Whitney received a large manufacturing contract to build muskets for the government. Whitney’s innovation was to divide the work so that each part was produced by a specialist to meet a specified standard. Each gunsmith focused on a single part, using sophisticated tools to optimize that task. The government, quickly realized that the standards would allow parts to be interchanged, greatly simplifying their firearm repair problems.

The importance of object-oriented programming is comparable to that of Whitney’s interchangeable part innovation, and for many of the same reasons. Both redefine the unit of modularity so that workers produce subcomponents instead of complete solutions. The subcomponents are controlled by standards and can be interchanged across different products.”

We can identify several basic features of object-oriented languages:

- encapsulation of state
- inheritance
- dynamic dispatch
- late binding (this)
- super
class A {
    private int x = 0;
}

class B extends A {
}

The code (new B()).x) does not compile.
A class can be instantiated to create new objects and subclassed to create new classes.

```java
class A {
    private int x = 0;
    int m() { x=x+1; return x; }
    int n() { x=x-2; return x; }
}

class B extends A {
    int o() { x=x*3; return x; }
}
```

The code `(new B()).m())` is legal as an instance of B does have a method `m`. We say the class B inherits the methods `m` and `n`, and the field `x`. 
Dynamic Dispatch

class/Dispatch.java
Dynamic Dispatch

class A {
    private int x = 0;
    int m() { x=x+1; return x; }
    int n() { x=x-2; return x; }
}

class B extends A {
    int m() { x=x*3; return x; }
}
class C extends A {
    int m() { x=x-7; return x; }
}

The (new B()).m()) invokes completely different code than The (new C()).m())).
### Subtype polymorphism

```java
interface A {
    int m();
}
class B implements A {
    private int i = 0;
    void m() { i=i+1; }
}
class C implements A {
    private float x = 0.0f;
    void m() { x=x+1.0f; }
}
class D {
    void p (A a) { a.m(); }
}
```

The method `p` can be invoked on objects of different classes, e.g., `(new D()).p(new B())` and `(new D()).p(new C())`, because of their interface is just as good as the interface required of the argument to `p`.

An interface sometimes implies just the set of operations defined on an object (as in Java), but the polymorphism extends to fields as well.
Late Binding

class A {
    private int x = 0;
    int m() { x=x-1; return x; }
    int n() { x=x+2; return this.m(); }
}

class B extends A {
    int m() { x=x*3; return x; }
}

Consider (new A()).n() and (new B()).n(); the first returns 1, the second 3.
```java
class A {
    private int x = 0;
    int m() { x=x-1; return x; }
    int n() { x=x+2; return this.m(); }
}

class B extends A {
    int m() { x=x*3; return super.m(); }
}

Consider \texttt{(new B()).n()}; 5 is returned.
```
Class: a group, set, or kind sharing common attributes
Instance: an individual or specimen with qualities or characteristics of the whole.
Object-Oriented Programming

Organize data structures into a hierarchy
Inheritance and dynamic dispatch in Java:

class/Shapes.java

// class/Shapes.java
Calling Procedure

- `call P(A)` – compiler looks up address of `P` and jumps to instruction
- `call P(A)` – (overloading) compiler chooses from among several procedures based on the static types of arguments
- `o.P()` – (dynamic dispatch) the runtime system chooses from among several procedures based on the subtype of object `o`

Note that static type checking is possible in all cases. Also, dynamic dispatch is not polymorphism—it is a strategy to resolve which of disparate functions (all of the same form) to call.
Narrowing

Coercions can be classified into those that preserve information (\textit{widenings}) and those that lose information (\textit{narrowings}). The coercion \texttt{int to long} is a widening; \texttt{int to short} is a narrowing. See the table of Java coercions at

\texttt{/ ryan/java/language/java-data.html}

The terms apply to the OO hierarchy as well. OO programming often requires narrowing which defeats the purpose of strong typing. See the Java program example:

\texttt{java/programs/class/Widening.java}
Dynamic Dispatch

class/inheritance.cc

class/Dispatch.java
class/dispatch.cc
Object-Oriented Programming

OO language use classes for encapsulation and subtyping. Ada, Modula-3, Oberon have modules and extensible records.

ada/programs/objects/shape_main.adb
Vtable
Subtyping Subtleties

Subtype polymorphism is not without its subtleties.

- Easy to confuse overloading with overriding.
  class/Override.java
- Favor aspects over inheritance
- Subtypes and arrays
Subtyping and Aspects

Equality, for example, does not mix well with subtypes.

java/programs/aspect/Point.java
java/programs/aspect/SubPoint.java
java/programs/aspect/Aspect.java

See *Effective Java Programming Language Guide* by Joshua Bloch. Chapter 3: Methods Common to All Objects.
Assignment and subtyping do not mix sometimes.

java/programs/misc/Sub.java
java/programs/misc/ASE.java
**Contravariance**

Vegetable <: Food  
Bat <: Mammal

```java
interface I {
    Mammal f (Vegetable v);
}

class C implements I {
    Bat f (Food d) { return new Bat(); }
}
```

Works in Java 1.5, but not Java 1.4: no contravariance.

java/programs/class/Contra.java  
java/programs/class/Vegetarian.java
Eiffel

Eiffel is not type-safe.

1. Replace result type by a subtype of the one occurring in the superclass
2. Hide inherited methods in a subclass

   A routine is a CAT (Changing Availability or Type) if some redefinition changes its export status or the type of one of its arguments. A call is a catcall if some redefinition of the routine would make it invalid because of a change of export status or argument type.

Not implemented, may not solve the problem, and may be so restrictive nothing interesting passes.
Mixin

The term “mixin” was first coined in the Flavors community [Weinreb81]. It has been used in a number of contexts; see, e.g., [Keene88], [Bracha90], [Booch94], [Taligent94], [VanHilst96]. In the context of multiple-inheritance languages like C++, a mixin is a partial class that implements a small part of the functionality of a large class (See [Booch94]). A mixin is distinguished from the other partial classes by the intent of its designers. A typical, partial class is like a mostly completed puzzle, with deferred methods representing a few missing pieces. A mixin is like a single puzzle piece, with final methods representing the tabs on a puzzle piece and deferred methods representing the indentations into which tabs of other pieces fit.

1. Chapter 6. Data Types
2. Chapter 9. Subprograms
3. Chapter 10. Implementing Subprograms
4. Chapter 11. Abstract Data Types
5. Chapter 12. Support for Object-Oriented Programming
6. Chapter 14. Exception Handling and Event Handling

Sebesta, seventh edition