Typing

Professor
Dharmendra Kumar Yadav

Data Types

- Hardware Data Types
- Virtual Data Types
- Abstract Data Types

Typing

- Typing is the enforcement by the class of an object, such that object of different types
  - May not be interchanged, or at the most
  - They may be interchanged only in very restricted ways
- Types derives from the theories of abstract data types
- A type is a precise characterization of structure or behavioral properties which a collection of entities all share
- We use the terms type and class interchangeably

Abstract Data Type (ADT)

- Type/Domain
- Functions/Operations
- Axioms
- Preconditions and Postconditions

An ADT Example: Unbounded Stack

Let E be the element type and T be Stack type.
T holds elements of type E.

The below operations are defined for this type.

T new (void)

T push (E,T)

E top(T)

T removetop(T)

Boolean empty (T)

Properties of the operations

- empty(new())

  new creates a nil stack

- top(push(e,t)) = e

pushed element goes on top, top gives the recently pushed element

- removetop(push(e,t)) = t

removetop retains the old stack prior to last push

- not empty(push(e,t))

when a push operation is performed, the stack becomes non empty

Partial Functions

- Some functions are not defined on all members of the input set

- Which of those defined above are partial functions?

Partial Functions in our Example

- *top* cannot return a value of type E for all values of input type T.
- Similarly *removetop* does not work on all values of input type T.
- How to handle the partially defined functions in ADT specification?

Preconditions of Partial Functions

- T removetop (T) requires not empty (T).
- E pop (T) requires not empty (T)

Summary of ADT Specification

- Types (used in the ADT)
- Functions (operations defined on these types)
- Axioms (properties over the functions defined)
- Preconditions

Observations

- Nowhere we used the notion of state
- Behavior was defined in terms of a set of pure functions and their properties
- It's not easy to generate an ADT specifications
- We can Convert ADT specifications into classes

Typing (Cont..)

Strong typing prevents mixing of abstractions

Typing (Cont..)

- Built-in (Primitive) Types
  - int
  - double
  - char
  - bool
  - ……
  - …..
  - …..

Typing (Cont..)

- User-Defined Types (UDT)
  - Complex
  - Vector
  - String
  - Employee
  - Executive
  - .....
  - .....

Typing of a Programming Language

A Programming language may be

- **Statically typed**-static binding or early binding
  - Types are associated with variables not values [C]

```
int    iVal=2;  // iVal is of type int
double dVal=3.6 // dVal is of type double
iVal = 3.6  // Implicit conversion of double --→ int
dVal=2      // Implicit conversion of int --→ double
```

  - Static type checking is the process of verifying the type safety of a program based on analysis of a program's text
  - Example: C, C++, JAVA, Ada

Typing of a Programming Language (Cont..)

- **Dynamically typed -** dynamic binding or late binding
  - Types are associated with values not variables [Python]

```
iVal=2;  // iVal is of type int
dVal=3.6 // dVal is of type double
iVal = 3.6  // iVal is of type double
dVal=2      // dVal is of type int
```

  - Dynamic type checking is the process of verifying the type safety of a program at run-time
  - Example: C++, JAVA(Polymorphism), Smalltalk

Typing of a Programming Language (Cont..)

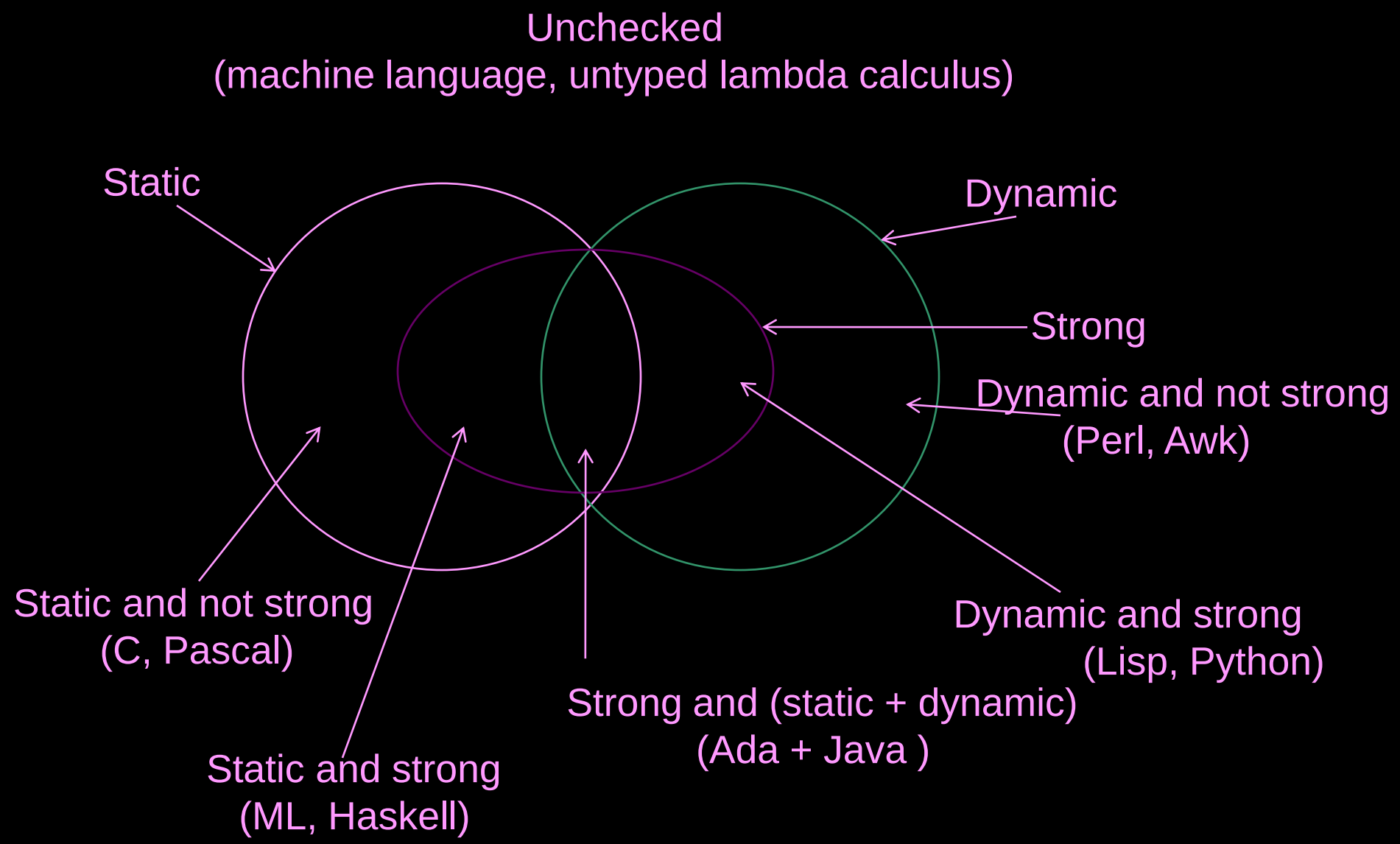
A Programming language may be

- **Strongly typed**
  - Typing errors are prevented at runtime
  - Allows little implicit type conversion
  - Does not use static type checking
  - Compiler does not check or enforce type constraint rules
  - Example Python, C++(void* of C), JAVA

Typing of a Programming Language (Cont..)

- **Weakly typed**
  - Easy to use a value of one type as if it were a value of another type
  - Allow implicit type conversion(in a type-safe manner)
  - Example: C, Ada, some feature of C++ and JAVA
- **Untyped**
  - There is no type checking
  - Any type conversion required is explicit
  - Example: Assembly language and Smalltalk

Typing (Cont..)


Unchecked
(machine language, untyped lambda calculus)
Static
Dynamic
Strong
Dynamic and not strong
(Perl, Awk)
Static and not strong
(C, Pascal)
Dynamic and strong
(Lisp, Python)
Strong and (static + dynamic)
(Ada + Java)
Static and strong
(ML, Haskell)

Polymorphism

- Polymorphism exists when dynamic typing and inheritance interact
- *A single name (Such as a variable) may denote objects of many different classes that are related by some common super class*
- *Monomorphism, in contrast, is supported by languages that are both strongly and statically typed*

Polymorphism is the most powerful feature of object-oriented programming languages next to the support for abstraction

Polymorphism (Cont..)

- Treats an object of derived classes as an object of base class
- so, we can write code that deals only with base classes
  - i.e. it deals with objects of type equivalent to base class: that includes all objects of derived class also!
- Extensible code: since our code deals only with base class, it can work with new datatypes (derived classes) that inherit from the base class

Polymorphism (Cont..)

- Polymorphism allows an entity (for e.g., variable, function or object) to take a variety of representations at different times
- Overloading of methods

```
int plus(int one, int two, int three);
int plus(int one, int two);
```

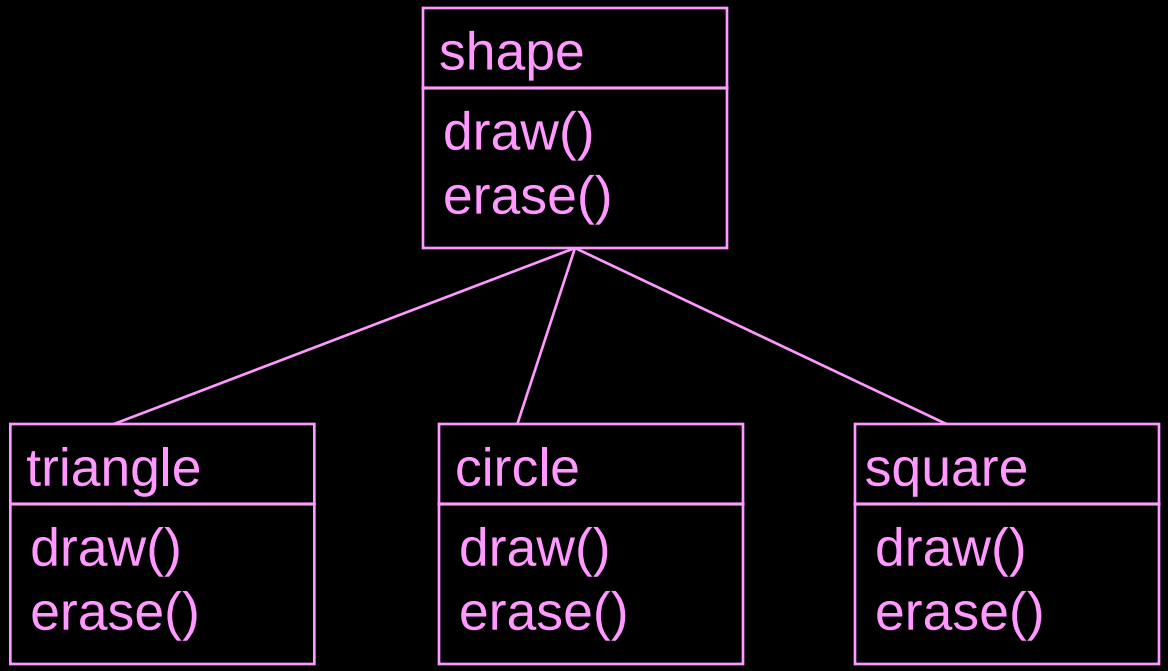
  - signature of method
- Objects of super-classes can be filled with objects of their sub-classes

```
public void printRec(Person per);
```

  - if *Male* and *Female* are sub-classes of class *Person*, they both can be passed to the method *printRec*

Interchangeable objects with Polymorphism

- Inheritance usually ends up creating a family of classes based on a uniform interface



Interchangeable objects with Polymorphism (Cont..)

- Example:

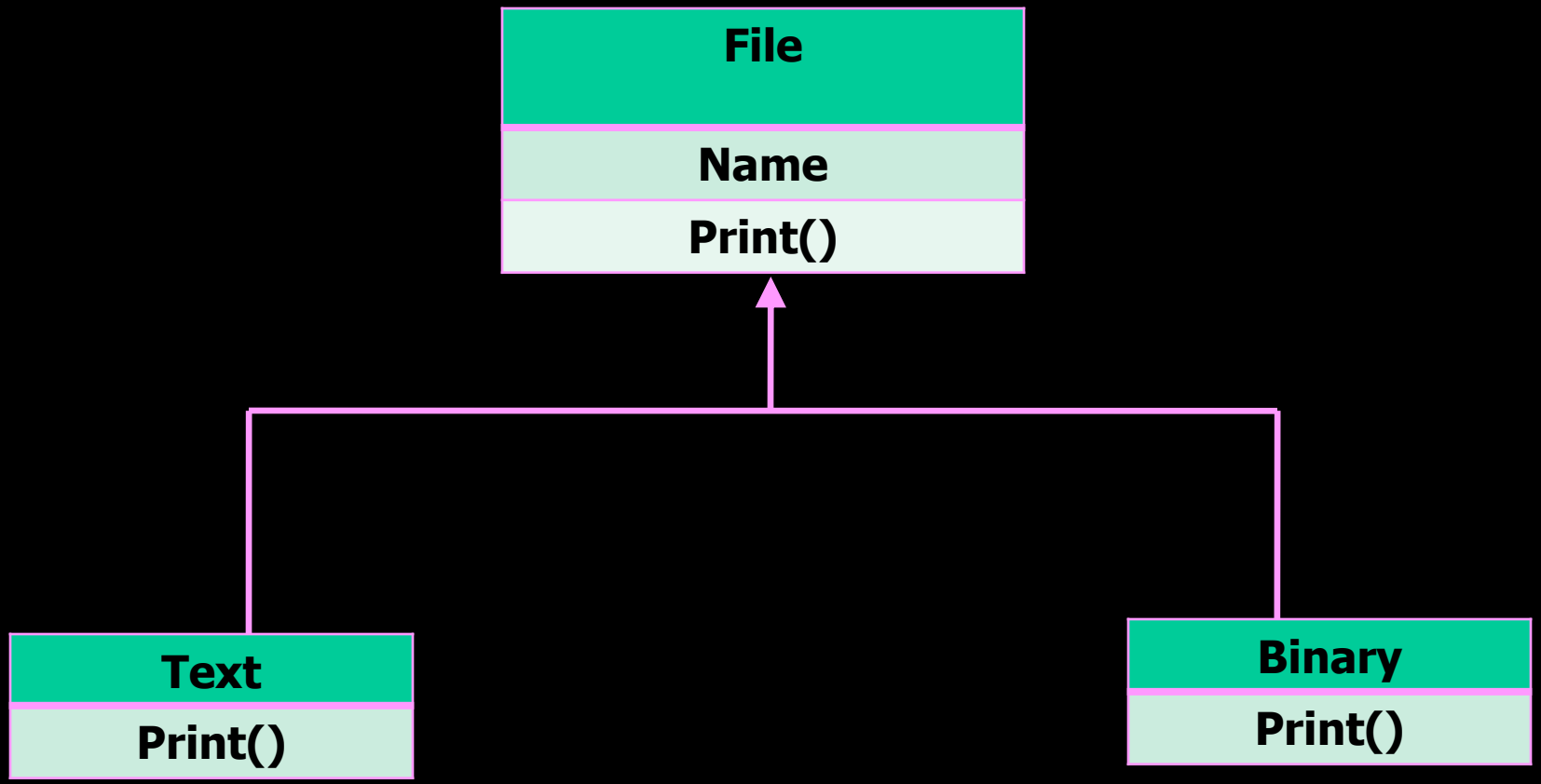
```
void doStuff(Shape s) {
s.erase();
//...
s.draw();
}

Circle c = new Circle();
Triangle t = new Triangle();
Line l = new Line();
doStuff(c);
doStuff(t);
doStuff(l);
```

Upcasting: can pass a circle or line object instead of shape object

Polymorphism implemented through dynamic binding

Interchangeable objects with Polymorphism (Cont..)

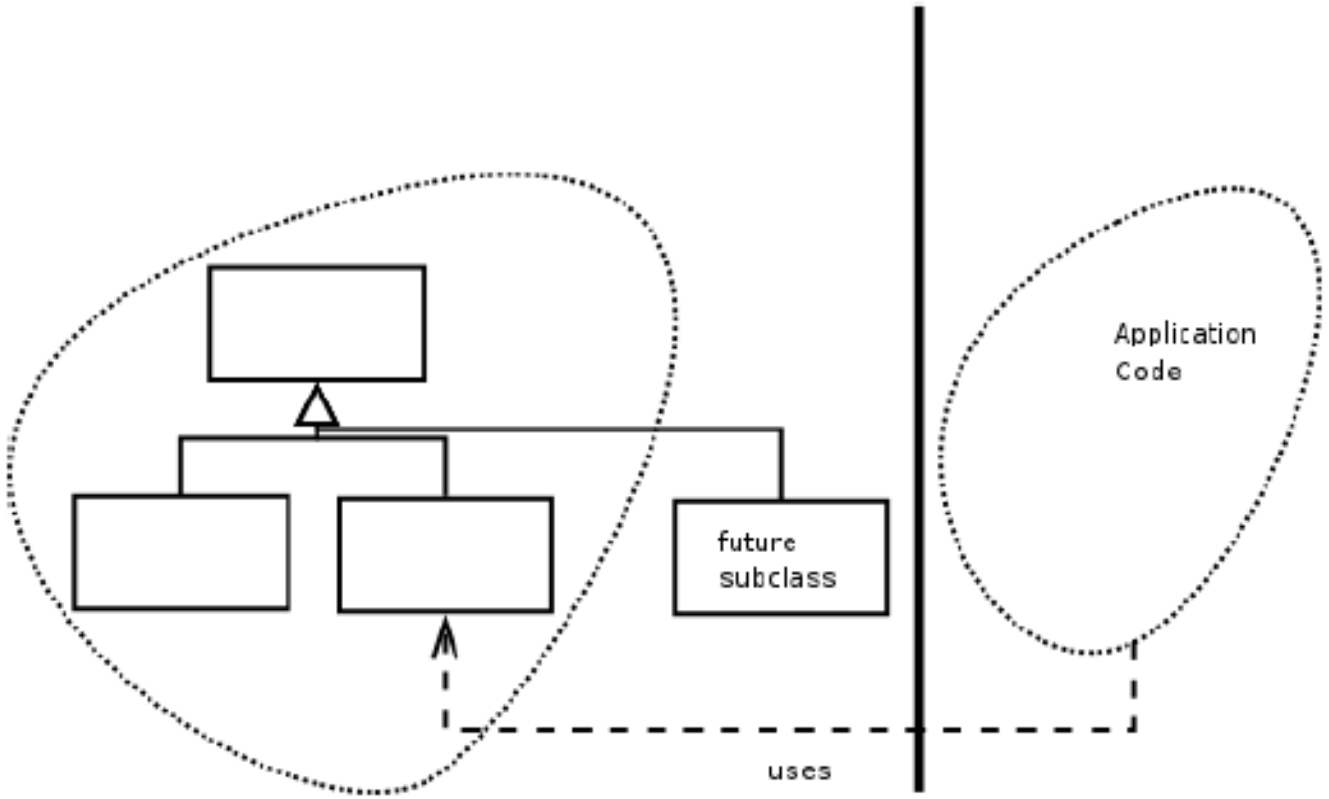


Interchangeable objects with Polymorphism (Cont..)

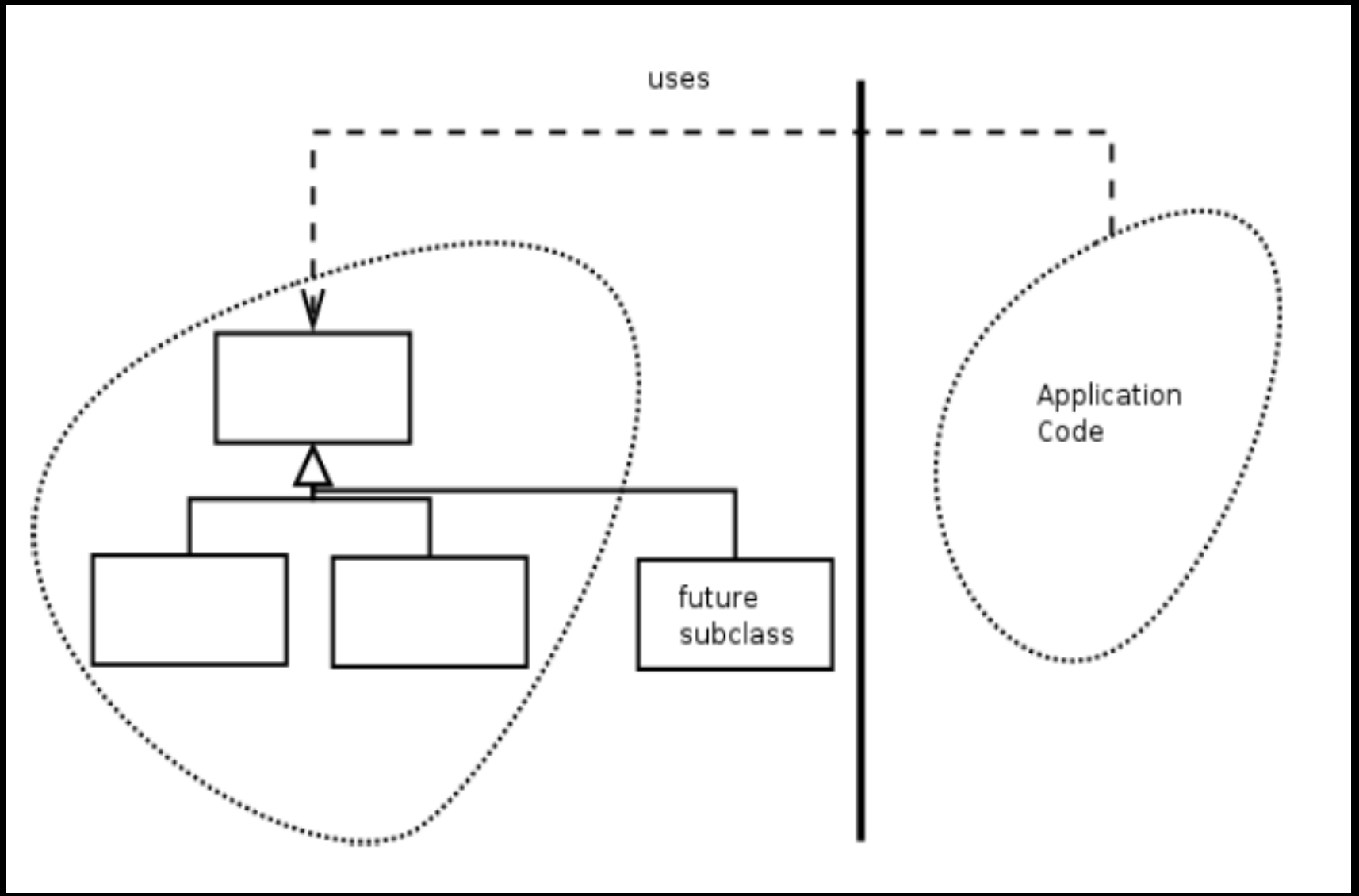
```
class client {
        public void handle file(file f){
         switch (f.gettype()) {
  case 'text':
      System.out.println("Text file");
  case 'Binary':
     System.out.println("Binary file");
         }

class client
   {
    public void do_it(file f);
    f.print();
     }
}
```

Reuse Through Extension and Refinements



Towards Higher Reuse through Polymorphism



Dynamic Binding and Polymorphism

```
class A {
        public  void f () { System.out.println( "A.f“); }
        public  void g () { System.out.println( "A.g”); }
        public  void h () { System.out.println( "A.h“);}
        public void k () { System.out.println( "A.k“) ;  }
}
class B extends  A {
        public void g () { System.out.println( "B.g“); }
        public void h () { System.out.println( "B.h“);  }
}
class C extends B {
        public void h () { System.out.println( "C.h");  }
        public  void k () { System.out.println("C.k“); };
}
}
```

Dynamic Binding and Polymorphism

```
public class db
{
public static void main(String args[]) {

        C cp = new C();  B bp = cp;
        A a1 = cp;
        A a2 = bp;
        A a3 = new B();
        cp.f(); cp.g(); cp.h(); cp.k();
        bp.f(); bp.g(); bp.h(); bp.k();
        a1.f(); a1.g(); a1.h(); a1.k();
        a2.f(); a2.g(); a2.h(); a2.k();
        a3.f(); a3.g(); a3.h(); a3.k();
}
```

Reusing the implementation

- Simplest way: just use an object of that class directly (not-very-often)
- placing an object of the class inside new class (part-of hierarchy)
- This is called "Composition". Compose a new class out of existing classes
- Inheritance
- Higher Reuse through Polymorphism