Subclass Instances

Constructors of the sub-class execute in three stages:

- Allocate an uninitialized instance of the class.
- Call the constructor of the super-class to initialize its fields.
- Initialize the sub-class by executing its constructor.

In Java you may write `super(args)` as the first statement of a sub-class constructor to initialize the super-class.

Classes and Instances

A class consists of three main parts:

- One or more constructors that create instances, or objects, of that class.
- Some instance variables, or fields, that are specific to each instance. Instance variables are initialized by a constructor of the class.
- Some methods that are shared among all instances of a class. The methods may access the instance variables.

Inheritance

Inheritance is the process of defining a new sub-class of a class by:

1. Defining new constructors for the class.
2. Adding new fields and/or methods to the super-class.
3. Overriding old methods in the superclass by replacing them with new ones.

Inheritance is sometimes called sub-classing. In Java the subclass is said to extend the super-class.

Single Inheritance

Java supports single inheritance.

- Every class has exactly one super-class which it is explicitly declared to extend.
- The class `Object` is the “top” of the inheritance hierarchy.

Single inheritance avoids ambiguity.

- What if there are two super-classes with a method `m`?
- What if one or more super-classes change?

Method Dispatch

Each instance is tagged with its class.

- The constructor applies the tag when the instance is created.
- The tag is often implemented as a pointer to the table of methods for the class.
Method Dispatch

Method dispatch is based on the class of the instance.

- The class determines which method to invoke.
- This is single dispatch, because the dispatch is based only on the instance, and not on the arguments of the method.

Visibility

The public components of a class describe each instance.

- Accessible fields and methods.
- Inherited by the sub-class.

Visibility

The private components are inaccessible, except to the methods of that class.

- Not part of the instance.
- Not inherited by the sub-class.

Visibility

The protected components are accessible to the sub-class.

- Not part of the instance.
- Inherited, subject to override.

(Protected components are also accessible to other classes in the same package.)

Inheritance and Subtyping

In Java the class of an object also serves as its type.

- It is important to distinguish the class, C, from the type, instanceof(C), of its instances.
- But it is conventional to blur this distinction!

Notation: A <: B for subtyping, A ⊑ B for inheritance.

Inheritance and Subtyping

In Java sub-classing implies sub-typing!

- If C inherits from D, then C is a sub-type of D. That is, if C ⊑ D, then C <: D.
- Therefore, by subsumption, we may provide a C instance wherever a D instance is required.
Inheritance and Subtyping

Why is it sound for sub-classing to imply sub-typing?
• Extension with new fields or methods yields a “fatter” object.
• Method override must respect the type of the method in the superclass!

Inheritance and Subtyping

Sub-classing is almost the only form of subtyping in Java.
• Base types such as int and float do not subtype.
• But arrays sub-type covariantly!
Array subtyping in Java violates subsumption!
• Fundamentally unsound.
• Imposes a run-time check on each assignment.

Method Override

Method override is powerful, but dangerous.
• Changes behavior of any methods that call the overridden method.
• Changes the semantics of every super-class!

Inheritance and Subtyping

Type of sub-class method must be a sub-type of the super-class method’s type.
• Can admit more general arguments.
• Can yield more specific results.

Java (prior to 1.5) required that the argument and result types be identical.

Important: knowing the type of an object does not determine its class!
• Might be an instance of any sub-class, by subsumption.
• The type is an attenuated record of the class of an object.

Method Override

Suppose D is a sub-class of C that overrides method m.
• Create a D instance, o.
• By subsumption, o is also a C instance.
• But the behavior of o may be arbitrarily different from that of a “true” C instance!
Inheritance and Behavior

Methodologically, if a class \( C \) extends a class \( D \), then the intention is that \( C \) specializes or enhances \( D \).

- The implementation of \( C \) is closely related to that of \( D \).
- A \( C \) instance exhibits \( D \) behavior in \( D \) contexts.

But it ain't necessarily so!

A sketch of a stack class:

```java
class Stack {
    protected Object[] Stack = null;
    private int top = 0;
    public void push (Object o) {
        Stack[top] = o;
    }
    public Object pop () {
        return Stack[top--];
    }
}
```

Inheritance and Behavior

A modification of Stack to exhibit queue-like behavior:

```java
class Queue extends Stack {
    private int bot = 0;
    public void push (Object o) {
        Stack[bot] = o;
    }
}
```

Now every Queue is a Stack!

This is a bad use of inheritance!

- A Queue instance does not behave like a Stack.
- The superclass behavior is not preserved under inheritance.

It is hard to distinguish good from bad uses of inheritance. (Perhaps they're all bad . . .)

Inheritance Is Anti-Modular

Inheritance is a relationship between implementations.

- Must inherit from a specific super-class, not any super-class of a given form.
- Reflects the accidents of implementation, rather than the essence of semantics.

Modularity relies on a separation between implementation and specification.

- Separable modules rely only on each other's interfaces.
- Reliance on implementation details violates modularity.
**Abstract Classes**

A class is **abstract** if the implementation of one or more methods is omitted.

- Abstract classes must be explicitly declared `abstract`.
- Abstract classes do **not** have instances.

Abstract classes force a sub-class to provide the missing behavior.

- Shared behavior provided by the super-class.
- Sub-class provides class-specific behavior.

**Inheritance Is Anti-Modular**

To override a method \( m \) of a class \( C \), you must understand the **implementation** of \( C \), not just its **specification**.

- Must understand which methods invoke \( m \) to understand how changing \( m \) affects the other methods.
- Cannot inherit “blindly” from a class without seeing the code.

This implies an **implementation-on-implementation** dependency, rather than an **implementation-on-interface** dependency.

**Controlling Inheritance**

Final classes are essentially ADT’s.

- An instance of a final class \( C \) **must** be implemented by that class and no other.
- Without `final` you cannot make such guarantees!

Compare ADT’s in ML: a value of an opaque type acts like a capability to perform the operations of the type and **no others**.

**But** ADT’s can have more than one opaque type in play at once, whereas objects get only one!

**Interfaces**

An interface is a **completely abstract** class.

- All methods are public, but unimplemented.
- All fields are `public static final`.

An interface is a description of behavior, not of implementation.

- Types of the methods.
- Informal semantics for the methods.
Interfaces and Subtyping

A class may be declared to implement one or more interfaces.

- Class must provide the specified methods with the specified types.
- An implementation should exhibit the behavior(s) of the interface(s).
- Often described as multiple inheritance of interfaces.

Multiple interface inheritance is OK because interfaces are completely abstract.

- There is no code to inherit from the super-classes!
- No ambiguity about which of several super-class methods of the same name is inherited.

Why are interfaces important?

- What you really want is subtyping, but all you have is inheritance.
- For flexibility you need multiple subtyping — a type can be a subtype of several types.

But you must explicitly declare the interfaces that a class implements.

- Must anticipate the future.
- Not robust under extensions to the system.

When to use interfaces?

- When writing “generic” code that only makes general behavioral requirements of an instance (e.g., in the “visitor pattern”).
- When using a general library that makes such requirements.

But what interfaces should you declare? Must anticipate all future uses!
Nominal and Structural Subtyping

Subtyping in Java is (almost entirely) nominal.

- Subtyping is induced only by inheritance (except for arrays).
- All subtyping relations among instance types must be explicitly declared.

Array subtyping is structural.

- Arrays subtype covariantly.
- Arrays are not classes.

Subsumption in Java

Implicit uses of subtyping:

- Arguments to methods or constructors can have more precise types than required by the method or constructor declaration.
- Method bodies can return values of a more precise type than specified in the result type of the method.

Subsumption in Java

Downcasts are often necessary to recover the “true” type of an instance.

- The compile-time type is attenuated by subsumption.
- Suppose \( a : A \) and \( A <: B \). If we “upcast” (explicitly or implicitly), we have \( a : B \). To recover the “true” type of \( A \) we “downcast” by writing \( (A)a \).

Subsumption in Java

A typical use of downcasting is to simulate polymorphism.

- Java didn’t (doesn’t?) have a type ‘a list.
- Instead we introduce a class List whose elements have type Object.
- Storing an element “forgets” its true type.
- On retrieval we downcast back to its true type.

It is up to the programmer to ensure that downcasts do not fail!
Subtyping Conditionals

What should be the typing rule for $e$? $e_1 : e_2$?

- Clearly $e$ must have type boolean.
- What should be required about $e_1$ and $e_2$?

Natural choice: $e_1$ and $e_2$ should have the same type $\tau$.

- **Without** subtyping, there is nothing else to say.

- **With** subtyping, we can allow $e_1$ and $e_2$ to have a common supertype, a type $\tau$ such that $e_1 : \tau_1 \leq \tau$ and $e_2 : \tau_2 \leq \tau$.

Subtyping Conditionals

Consider these interface declarations:

```java
interface A
interface B
interface C extends A, B
interface D extends A, B
```

(Multiple extension is OK for interfaces.)

It is easy to see that $C$ and $D$ have no least upper bound — both $A$ and $B$ are incomparable upper bounds.

Subtyping Conditionals

Natural choice: $\tau$ is the least common supertype (least upper bound) of $\tau_1$ and $\tau_2$.

- Avoids being too conservative, e.g. $\tau = \text{Object}$.
- Ensures a unique choice of type for conditionals.

Unfortunately least upper bounds don’t exist in Java!

Subtyping Conditionals

Java imposes an ad hoc rule to cover this case:

either $\tau_1 \leq \tau_2$ or $\tau_2 \leq \tau_1$, otherwise error.

It “works”, but it also uncovers a flaw in the design — lub’s are required, but do not necessarily exist.

Three Roles for a Class

A class has three conceptually distinct roles in Java:

1. Locus of code sharing and inheritance.
2. Compile-time type of instances of that class.
3. Run-time tag of instances of that class.