JAVA

As in Featherweight Java, classes in Java are a collection of
- fields, and
- methods

In Java, fields and methods can be declared to be
- dynamic (default)
  - per instance
- static
  - per class

Visibility can be controlled by declaring fields and methods to be
- public:
  - accessible by any client
- protected:
  - accessible only by instances of subclasses
- private:
  - not accessible by other instances
- final:
  - read-only, cannot be changed after initialization

TYPES

All components of a class have types.

The type of
- a field specifies the type of the values it can be assigned to
- a method specifies the number and type of its arguments, and the type of its result
- a constructor specifies the number and type of its arguments.
  The result type of a constructor is always the instance type of the class itself

SUBCLASSES

Classes can inherit the visible fields and method of another class.
- Java supports only single inheritance.
- A class has at most one direct superclass

Two types of inheritance:
- Enrichment
  - subclass provides additional methods and fields
- Overriding
  - subclass redefines method of superclass
  - old version can still be accessed via pseudo-variable super
Controlling inheritance:
- private and final methods cannot be redefined by subclass
- final methods can change indirectly, if they invoke non-final methods of class
- if a class is declared final, no other class can inherit from this class

Abstract Classes and Interfaces
- classes with methods which are only declared and not implemented are called abstract classes
- methods of the abstract class can invoke abstract methods
- abstract classes do not have instances
- fully abstract classes

Interfaces are fully abstract classes:
- all its fields are public static final — constants
- all its methods are abstract public

A class may be declared to implement one or more interfaces.
Therefore, Java is said to have
- single inheritance of implementation
- multiple inheritance of interfaces

Types in Java
1. **Base Types**: types like int, float, boolean as similar types in other languages
2. **Class Types**: which classify the instances of a class
   - “class” refers both to the program structure, and the type of its instances (also called the instance type)
3. **Array Types**: of the form τ[], which represents a mutable array of element type τ
   - Java arrays offer operations to create and initialise arrays, run bound checks
**Slide 9**

**Subtyping**

- $C$ is a subclass of $C'$: $C \triangleleft C'$
- Reflexive and transitive closure of the `extends` relationship
- The class $C$ implements the interface $I$: $C \triangleright I$
- Note that a class can implement more than one interface

**Slide 10**

Reflexivity and Transitivity:

$$\tau \triangleleft \tau$$

Covariance of Arrays:

$$\tau \triangleleft \tau' \quad \Rightarrow \quad [\tau] \triangleleft [\tau']$$

Subclasses and Implementations:

$$C \triangleleft C' \quad C \triangleleft I \quad C <: C' \quad C <: I$$

Special Class `Object`:

$$\tau \triangleleft \text{Object}$$

**Slide 11**

**Subsumption**

- If $\tau' \triangleleft \tau$, then every expression of type $\tau'$ is also an expression of type $\tau$.
- If a method or constructor is declared to accept a parameter of type $\tau$, we can also provide an expression of type $\tau'$.
- If a method is declared to return a value of type $\tau$, it is also possible to return a value of type $\tau'$.

**Slide 12**

What is the type of the conditional $e$? $e_1 : e_2$

- $e$ should have type `bool`.
- $e_1$ and $e_2$ should have the same type.
- What does this mean in the presence of subtyping?

- Requiring types to be exactly the same would be too restrictive.
- Is it possible to use the least upperbound?
- Not every pair of types has a least upperbound.
- Java requires one of the types to be a subtype of the other.
Arrays and Subtyping

- Arrays are covariant in Java
- Operations on arrays
  - retrieve elements
  - update elements

This leads to a problematic situation

Example:
- Let A be an array of colorpoints
- By subsumption, A is also an array of points
- Retrieving an element yields a value of type colorpoint, which by subsumption is also of type point
- As A is an array of points, we can assign a value of type point to an element of A
- But now, A is not an array of colorpoints any more!

This means Java would not be type safe!
Java solves this by applying expensive run-time checks whenever array values are updated
Not for arrays of int and float

Dynamic Dispatch

In Java, if
- C is a subclass of D, then
- C is a subtype of D

Therefore,
- If the static type of an expression is type D, it might in reality, be of type C
- The static type of an expression is just an approximation of the actual, dynamic type

In particular,
- Method dispatch is based on the dynamic type, not the static type
Static type determines
- which method names are visible
- which field names are visible

Dynamic type determines
- which method is invoked
- which casts are valid

Casts
- Up casts:
  - static type is subclass of target type
- Down casts:
  - static type is superclass of target type. Run-time checks required
- Invalid casts:
  - static type conflicts with target type, rejected by compiler

In languages like Haskell, when defining a collection data structure where neither the structure nor the operations depend on the actual elements, we use polymorphism.

In Java (up to 1.4), we can use the class `Object` since every class is a subclass of `Object`, elements of any subclass can be put into the container.

However, when retrieving an element, the (static) type information is lost.

No operation can be applied to element, as method dispatch and field selection checked statically.

downcast necessary

```java
class A {
}

class B {
}

Vector v = new Vector();

v.add(new A());

b B = (B) v.get(0);
```
**Generics in Java 1.5**

```java
Vector<A> v = new Vector<A>();
v.add(new A());
b = (B)v.get(0);
```

Compile time error:

inconvertible types
found : A
required: B

```java
B b = (B)v.get(0);
```

**Wildcards**

```java
Vector? extends Mammal> pets = dogs;
```

Similar to Haskell type:

```latex
\textit{pets : Mammal T} \Rightarrow \textit{Vector T}
```