As in Featherweight Java, classes in Java are a collection of

- fields, and
- methods
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)
- static
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:

- protected:

- private:

- final:
Visibility can be controlled by declaring fields and methods to be

- public:
  - accessible by any client

- protected:

- private:

- final:
Visibility can be controlled by declaring fields and methods to be

- **public:**
  - accessible by any client

- **protected:**
  - accessible only by instances of subclasses

- **private:**

- **final:**
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:**
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.
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 $\tau[\ ]$, which represents a mutable array of element type $\tau$
   Java arrays offer operations to create and initialise arrays, run bound checks
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
Reflexivity and Transitivity:

\[
\begin{align*}
\tau & \ll \tau \\
\tau_1 & \ll \tau_2 \\
\tau_2 & \ll \tau_3 \\
\tau_1 & \ll \tau_3
\end{align*}
\]

Covariance of Arrays:

\[
\begin{align*}
\tau & \ll \tau' \\
\tau[\ ] & \ll \tau'[\ ]
\end{align*}
\]

Subclasses and Implementations:

\[
\begin{align*}
C \ll C' & \quad C \ll I \\
C & \ll C' \\
C & \ll I
\end{align*}
\]

Special Class `Object`:

\[
\tau \ll \text{Object}
\]
if $\tau' <: \tau$, 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 $\tau'$
What is the type of the conditional \( e? e_1 : e_2 \)
What is the type of the conditional $e \, ? \, e_1 \, : \, e_2$

→ $e$ should have type $bool$
What is the type of the conditional $e? e_1 : e_2$

$\rightarrow$ $e$ should have type $bool$

$\rightarrow$ $e_1$ and $e_2$ should have the same type
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?
What is the type of the conditional $e? \; e_1 : e_2$

$\rightarrow$ $e$ should have type $\textit{bool}$

$\rightarrow$ $e_1$ and $e_2$ should have the same type

$\rightarrow$ what does this mean in the presence of subtyping?

- requiring types to be exactly the same would be too restrictive
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?
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
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
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
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

① Up casts:
   - static type is subclass of target type
**Casts**

1. **Up casts:**
   - static type is subclass of target type

2. **Down casts:**
   - static type is superclass of target type. Run-time checks required
CASTS

① Up casts:
   - static type is subclass of target type

② Down casts:
   - static type is superclass of target type. Run-time checks required

③ Invalid casts:
   - static type conflicts with target type, rejected by compiler
Casts

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.
Casts

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`
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
class A {
}

class B {
}

Vector v = new Vector();

v.add(new A());

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

Vector <A> v = new Vector<A> ();

v.add (new A ());

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

Vector <A> v = new Vector<A> ();

v.add (new A());

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

Compile time error:

inconvertible types
found    : A
required: B
        B b = (B) v.get(0);
Vector<String> v2 = new Vector<String>();
Vector<Object> v = v2;
Vector<String> v2 = new Vector<String>();
Vector<Object> v = v2;

v.add(new A());
String s = v2.get(0);
WILDCARDS

Vector<? extends Mammal> pets = dogs;
Wildcards

Vector<? extends Mammal> pets = dogs;

similar to Haskell type:

\[ pets :: Mammal \ T \Rightarrow Vector \ T \]