

Covariance

- **Covariance** Definizione
- T si dice covariante (rispetto alla sottotipazione di Java) se ogni volta che A è sottotipo di B allora anche T di A è sottotipo di T B
 - T potrebbe essere il valore ritornato
 - ...

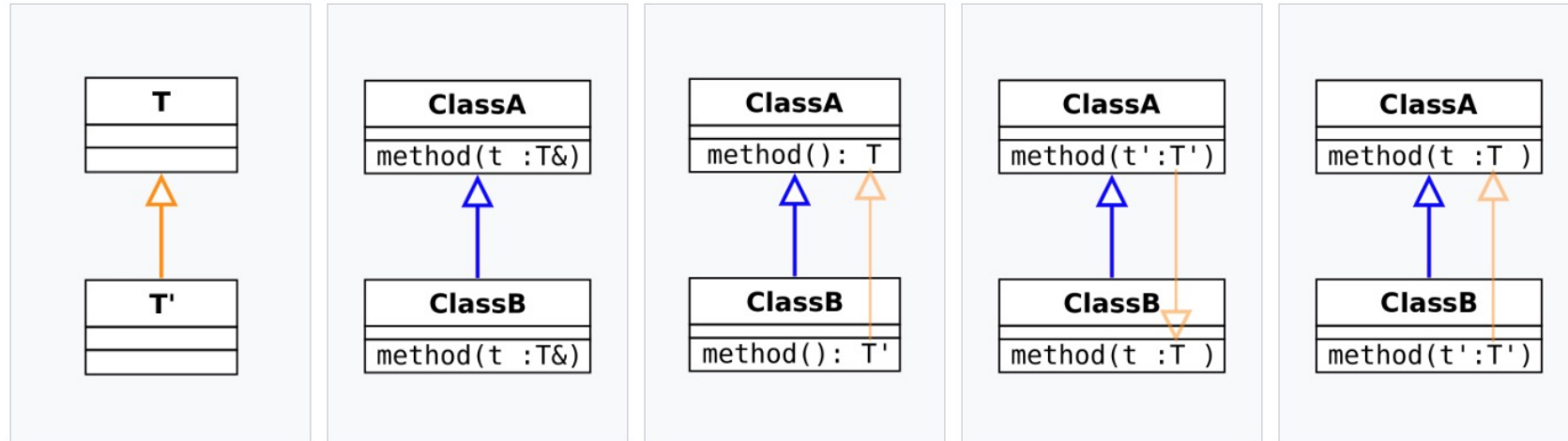
Liskov substitution principle (LSP) -1987

Subtype Requirement: Let $\phi(x)$ be a property provable about objects x of type T . Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T .

Symbolically:

$$S \leq T \rightarrow \forall x:T. \phi(x) \rightarrow \forall y:S. \phi(y)$$

Variance and method overriding: overview



Subtyping of the parameter/return type of the method.

Invariance. The signature of the overriding method is unchanged.

Covariant return type. The subtyping relation is in the same direction as the relation between ClassA and ClassB.

Contravariant parameter type. The subtyping relation is in the opposite direction to the relation between ClassA and ClassB.

Covariant parameter type. Not type safe.

Covariance

- **Covariance** in Java 5
- I valori ritornati da un metodo ridefinito possono essere covarianti
- parameter types have to be exactly the same (invariant) for method overriding, otherwise the method is overloaded with a parallel definition instead.

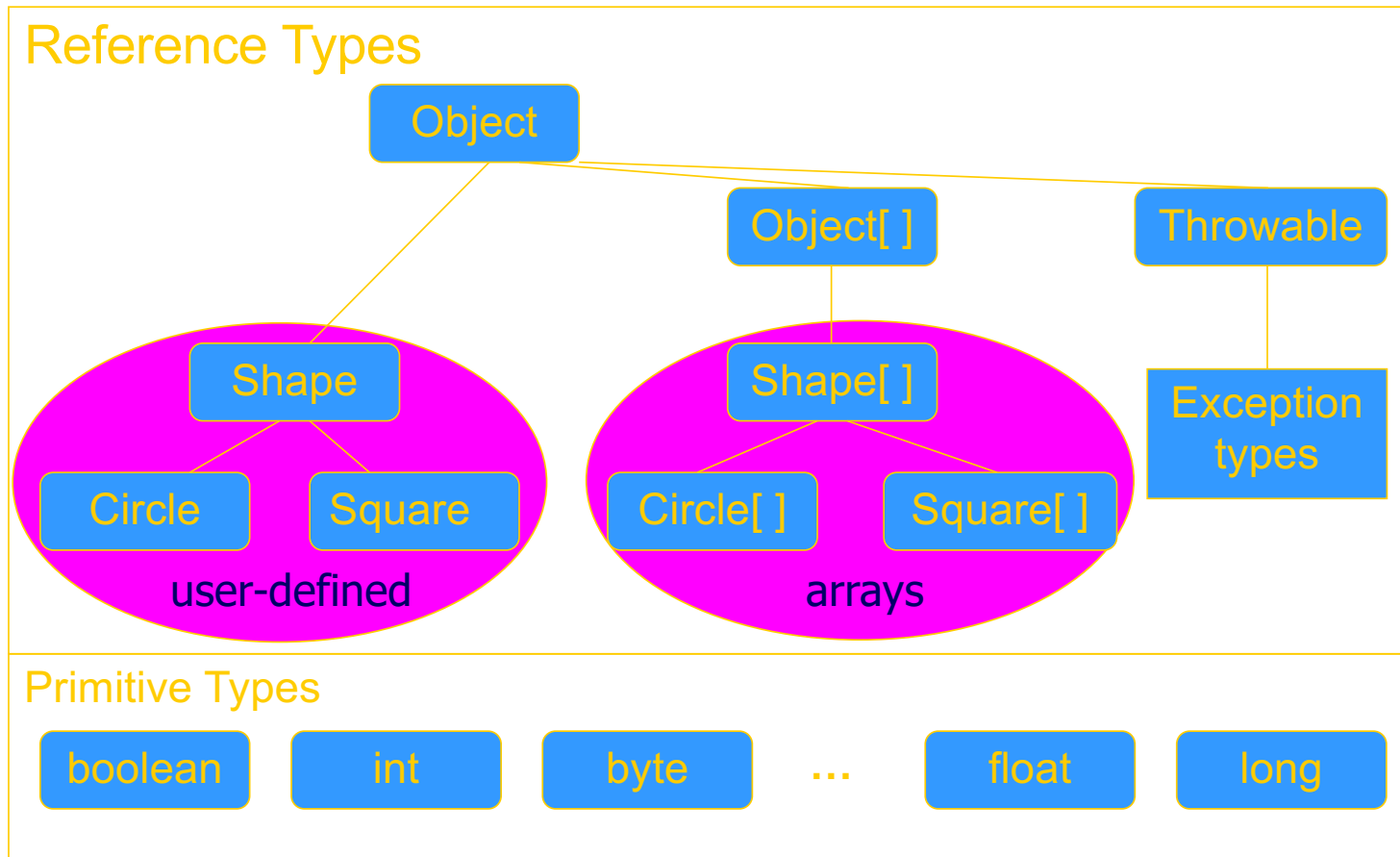
```
class A {  
    public A whoAreYou() {...}  
}  
  
class B extends A {  
    // override A.whoAreYou *and* narrow the return type.  
    public B whoAreYou() {...}  
}
```

Java

Array types

- Automatically defined
 - Array type `T[]` exists for each class, interface type `T`
 - Cannot extended array types (array types are final)
 - Multi-dimensional arrays as arrays of arrays: `T[][]`
- Treated as reference type
 - An array variable is a pointer to an array, can be null
 - Example: `Circle[] x = new Circle[array_size]`
 - Anonymous array expression: `new int[] {1,2,3, ... 10}`
- Every array type is a subtype of `Object[]`, `Object`
 - Length of array is not part of its static type

Classification of Java types



Array subtyping - covariance

- Covariance

- if $S <: T$ then $S[] <: T[]$
 - $S <: T$ means “S is subtype of T”

- Standard type error

```
class A {...}
class B extends A {...}
B[] bArray = new B[10]
A[] aArray = bArray // considered OK since B[] <: A[]
aArray[0] = new A() // compiles, but run-time error
                    // raises ArrayStoreException
// b/c aArray actually refers to an array of B objects
// so that assignment, aArray[0] = new A(); would violate the type of bArray
```

Java Generic Programming

- Java has class Object
 - Supertype of all object types
 - This allows “subtype polymorphism”
 - Can apply operation on class T to any subclass $S \leq T$
- Java 1.0 – 1.4 do not have templates
 - No parametric polymorphism
 - Many consider this the biggest deficiency of Java
- Java type system does not let you cheat
 - Can cast from supertype to subtype
 - Cast is checked at run time

Why no generics in early Java ?

- Many proposals
- Basic language goals seem clear
- Details take some effort to work out
 - Exact typing constraints
 - Implementation
 - Existing virtual machine?
 - Additional bytecodes?
 - Duplicate code for each instance?
 - Use same code (with casts) for all instances

Java Community proposal (JSR 14) incorporated into Java 1.5

Motivazione per l'introduzione dei generici

- **Programmazione generica**
- Se voglio realizzare programmi generici, cioè che vanno bene per diversi tipi, come posso fare?
- Posso usare scrivere gli algoritmi usando Object che a runtime potrà essere una qualsiasi sottoclasse
- Così era prima di 1.5
- Ad esempio una collezione generica

Esempio Lista di Object

(prima dei generici)

Ad esempio una lista

```
// creazione
List myList = new LinkedList();

// aggiungo
myList.add(new Integer(0));

// prendo il primo elemento
Integer x = (Integer)
    myIntList.iterator().next();
```

1. Il cast è necessario
2. posso inserire qualsiasi oggetto

Stack:

```
class Stack {
    void push(Object o) {...}
    Object pop() { ... }
    ...}
```

```
String s = "Hello";
Stack st = new Stack();
...
st.push(s);
...
s = (String) st.pop();
```

Come specializzare (senza generics)

- Posso specializzare mediante ereditarietà (senza usare i generics)
- Ad esempio se voglio una lista che prende solo gli interi

```
IntegerList extends ArrayList{  
    @Override  
    // devo mantenere la segnatura  
    boolean add(Object o){  
        // check o is Integer ...  
    }  
    @Override  
    //posso specializzare il return (cov.)  
    Integer get(int i){...}  
}
```

Sintassi dei generici

- Una versione generica della classe `Stack`
- Una classe generica è definita con il seguente formato:

```
class ClassName <T1 , T2 , ..., Tn > { ... }
```

- La sezione dei parametri di tipo, delimitati da parentesi angolari (<>), segue il nome della classe. Essa specifica i parametri di tipo (chiamati anche **variabili di tipo**) T1, T2, . . . e Tn.

Generics

Invece mediante i generici:

```
class Stack<A> {  
    void push(A a) { ... }  
    A pop() { ... }  
    ...  
}  
  
String s = "Hello";  
Stack<String> st = new Stack<String>();  
st.push(s);  
...  
s = st.pop();
```

Annoto con \diamond il TIPO
Generico. A **non** è una classe

Declaring Generic classes

- For example a Coppia of two objects one of type E and the other of type F

```
class Coppia<E,F>{  
    E sinistro;  
    F destro;  
  
    Coppia(E a, F b) { ... }  
  
    E getSinistro() { return sinistro; }  
  
}
```

Metodi generici

- *Generic methods* are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Static and non-static generic methods are allowed, as well as generic class constructors

```
public class Util {  
    public static <K,V> boolean compare(Pair<K,V> p1, Pair<K,V> p2)  
    {  
        return p1.getKey().equals(p2.getKey()) &&  
            p1.getValue().equals(p2.getValue());  
    }  
}
```


Bounded Type Parameters

Constraints on generic types

- One can introduce constraints over a type used as parameter in a generic class

`< E extends T> : E must be a subtype of T`

`< E super T> : E must be a supertype of T`

Esempio:

```
public <U extends Number> void inspect(U u){  
    System.out.println("U: " + u.getClass().getName());  
}
```

Metodi generici

- Analogamente a classi e interfacce generiche, in Java 5.0 è possibile definire metodi generici, ovvero parametrici rispetto ad uno o più tipi.

```
public class MaxGenerico {  
    public static <T extends Comparable<T>>  
        T max (Vector<T> elenco) {  
        ...  
    }  
}
```

- Nell'esempio:
 - la classe non ha parametri di tipo;
 - la dichiarazione di tipo è <T extends Comparable<T>>, immediatamente successiva ai modificatori;
 - il tipo del metodo è T;
 - la segnatura del metodo è max(Vector<T>).

Java generics are type checked

- A generic class may use operations on objects of a parameter type
 - Example: `PriorityQueue<T> ... if x.less(y) then ...`
- Two possible solutions
 - C++: Link and see if all operations can be resolved
 - Java: Type check and compile generics w/o linking
 - This requires programmer to give information about type parameter
 - Example: `PriorityQueue<T extends ...>` —

Example: Hash Table

```
interface Hashable {  
    int    hashCode ();  
};
```

This expression must typecheck
Use "Key extends Hashable"

```
class HashTable < Key extends Hashable, Value> {  
    void    insert (Key k, Value v) {  
        int bucket = k.hashCode();  
        insertAt (bucket, k, v);  
    }  
    ...  
};
```

Interface Comparable<T>

- imposes a total ordering on the objects of each class that implements it (natural ordering)
- **int compareTo(T o):** comparison method
 - compares **this** object with o and returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
- Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort).
- Objects that implement this interface can be used as keys in a sorted map or elements in a sorted set, without the need to specify a comparator.

compareTo

- The natural ordering for a class C is said to be **consistent** with equals if and only if $(e1.compareTo((Object)e2) == 0)$ has the same boolean value as $e1.equals((Object)e2)$ for every e1 and e2 of class C.
- Altri vincoli:
 - $sgn(x.compareTo(y)) == -sgn(y.compareTo(x))$
 - the relation must be transitive:
 - $(x.compareTo(y) > 0 \ \&\& \ y.compareTo(z) > 0)$ implies $x.compareTo(z) > 0$.
 - Finally, the implementer must ensure that $x.compareTo(y) == 0$ implies that $sgn(x.compareTo(z)) == sgn(y.compareTo(z))$, for all z.


Example

Class MyClass implements

```
Comparable<MyClass>{  
    private int a;  
  
    ...  
  
    public int compareTo(MyClass  
other){  
        return (this.a - other.a);  
    }  
}
```

Priority Queue Example

Generic types often requests the implementation of Comparable:



```
class PriorityQueue<T> extends Comparable<T> {  
    List<T> queue;    ...  
    void insert(T t) {  
        ... if (t.compareTo(queue.get(i))  
        ...  
    }  
    T remove() { ... }  
    ...  
}
```


No covarianza dei generics → conseguenze

- Nota che se S è sottotipo di T una classe P<S> non è sottotipo di P<T>
 - In questo modo non ho i problemi degli array
- A è sottotipo di Object
- Collection<A> non è Collection<Object>

Generics and Subtyping

- Questo è corretto?

```
1. List<String> ls = new ArrayList<String>();
```

```
2. List<Object> lo = ls;
```

- 1 sì (arrayList è un sottotipo di List).
- Ma 2? Una Lista di String è un sottotipo di una stringa di Object
- Attenzione, se fosse vero avrei ancora problemi simili a quelli degli array

```
lo.add(new Object()); // 3
```

```
String s = ls.get(0); // 4: attempts to assign an Object to a String!
```

- **NON C'è covarianza dei generici**
- **A <: B non implica I<A> sottotipo di I !!**

Generics e wildcard

- Vogliamo scrivere un metodo che prende una collezione e stampa tutti gli elementi:

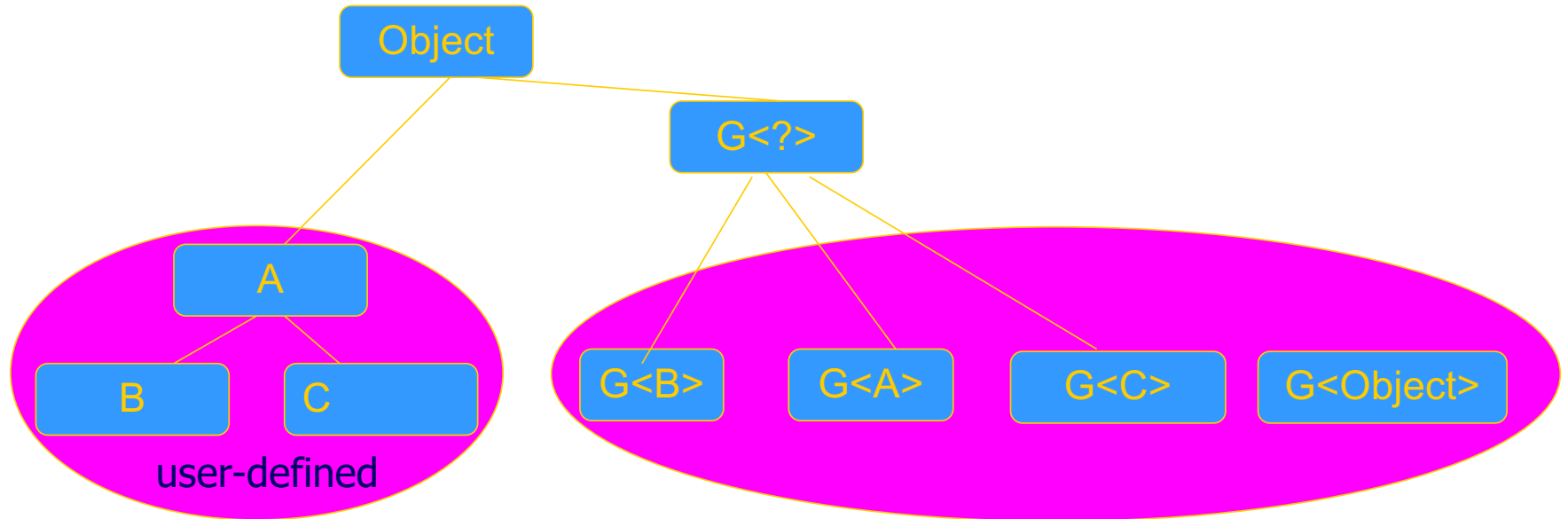
```
void printCollection(Collection c) {...}
```

- Con i generics???

```
void printCollection(Collection<Object> c) {  
    Iterator i = c.iterator();  
    for (k = 0; k < c.size(); k++) {  
        System.out.println(i.next().toString());  
    }  
}
```

- E se ho Collection<Student> non funziona !!!
- C'è un supertipo di Collection<Student>, Collection<...> ...??

Supertipo di generics



```
void printCollection(Collection<?> c) {  
    for (Object e : c) {  
        System.out.println(e);  
    }  
}
```

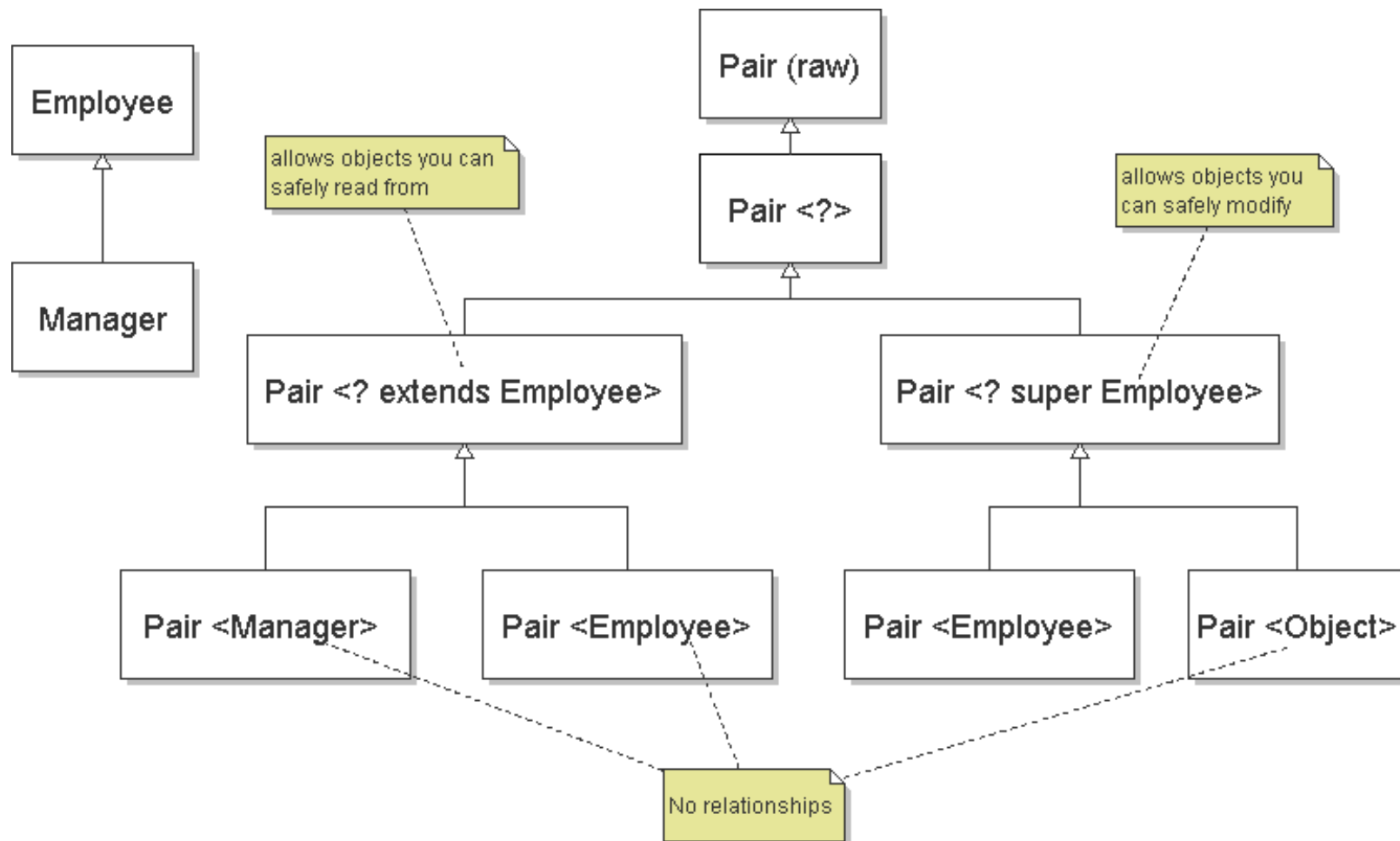
Upper Bound su wildcard generics

- Esempio:
Studente <: Persona, non ho che
List<Studente> <: List<Persona>
- Esempio:
 - stampaAnagrafica(List<Persona> p)
 - List<Studente> ls;
 - **stampaAnagrafica(ls)** non compila
 - Non posso però neanche definire:
 - stampaAnagrafica(List<?> p)
 - Esiste un tipo intermedio: List<? extends Persona>

Wildcard e generics (Lower bound)

- Alcune volte non si vuole specificare esattamente il tipo ma si vuole essere più permissivi
- `Persona` **extends** `Comparable<Persona>`
- `Studiante` **extends** `Persona`
- `Studiante` non può essere sostituito a `T` in un generico che chiede `<T extends Comparable<T>>`
 - Non potrei fare liste ordinate di studente
 - Però potrei utilizzare il `compareTo` di `Persona`, senza necessità di introdurre un altro `compareTo` nella sottoclasse
- Introduco: `<T extends Comparable<? super T>`

Inheritance rules for generic types



Comments on inheritance relations

- `Pair<Manager>` matches `Pair<? extends Employee>` => subtype relation (*covariant* typing)
- `Pair<Object>` matches `Pair<? super Employee>` => subtype relation (*contravariant* typing)
- `Pair<Employee>` can contain only *Employees*, but `Pair<Object>` may be *assigned* anything (*Numbers*) => *no* subtype relation
- also: `Pair<T> <= Pair<?> <= Pair (raw)`

```
List <String> sl = new LinkedList <String> ();  
List x = sl;           // OK  
x.add (new Integer (5)); // type safety warning  
String str = sl.get (0); // throws ClassCast.
```


Implementing Generics

- Type erasure
 - Compile-time type checking uses generics
 - Compiler eliminates generics by erasing them
 - Compile `List<T>` to `List`, `T` to `Object`, insert casts
- “Generics are not templates”
 - Generic declarations are typechecked
 - Generics are compiled once and for all
 - No instantiation
 - No “code bloat”

More later when we talk about virtual machine ...

Esercizio

- Dichiarare una classe A che ha come membro un intero
 - Dichiarare una classe B extends A che ha un metodo equals(B a)
 - Dichiarare una classe C extends A che ha un metodo equals(Object)
 - Implementare i metodi toString in modo che stampino "A", "B" e "C" e il valore dell'intero
 - Dichiarare una Lista di A usando i generici
 - Inserisci qualche B e qualche C
 - Stampa il contenuto della lista con un ciclo for each
 - Domanda un intero x
 - `Scanner sc = new Scanner(System.in);`
 - `int x = sc.nextInt();`
 - e cerca nella lista un elemento che sia equals a new A(x)
 - usa for each e equals
 - usa contains
- QUALI PROBLEMI HAI???

Auto boxing /unboxing

- Adds auto boxing/unboxing

User conversion

```
Stack<Integer> st =  
    new Stack<Integer>();  
st.push(new Integer(12));  
...  
int i = (st.pop()).intValue();
```

Automatic conversion

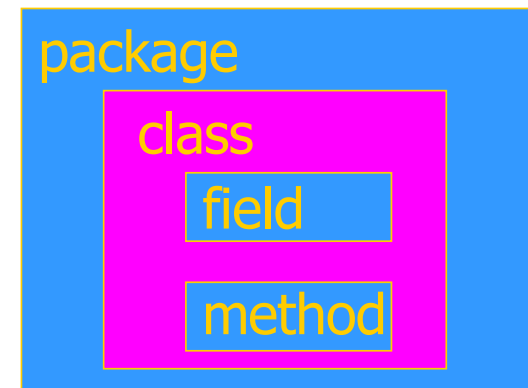
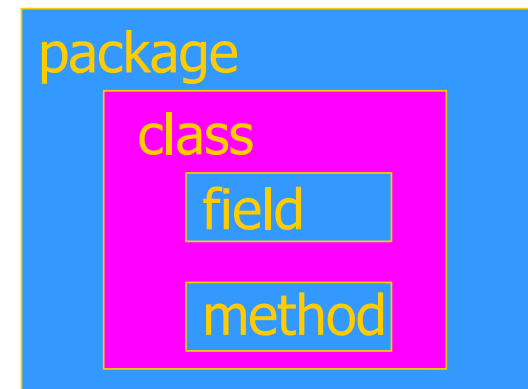
```
Stack<Integer> st =  
    new Stack<Integer>();  
st.push(12);  
...  
int i = st.pop();
```

Package e visibilità (18)

Packages and visibility

Encapsulation and packages

- Every field, method belongs to a class
- Every class is part of some package
 - Can be unnamed default package
 - File declares which package code belongs to



Visibility and access

- Four visibility distinctions
 - public, private, protected, package
- Method can refer to
 - private members of class it belongs to
 - non-private members of all classes in same package
 - protected members of superclasses (in diff package)
 - public members of classes in visible packages

Visibility determined by files system, etc. (outside language)
- Qualified names (or use import)
 - `java.lang.String.substring()`

 `package` `class` `method` java

Visibilità e overriding

- Quando si ridefinisce un metodo, questo non deve essere privato, altrimenti si fa overloading.

Esempio

```
class A {  
    private void m(String s)  
        { /* ... */  
    }  
    void m(Object o) { /* ... */ }  
}  
  
class B extends A {  
    void m(String s) { /* ... */ }  
}
```

La classe B non ridefinisce m di A ma fa overloading:

Main in altra classe

```
A a = new A();  
a.m("def"); ---  
m(object)  
  
A b = new B();  
b.m("def"); - idem
```

Overriding e visibilità

- Quando si ridefinisce, la visibilità può solo aumentare.
- Esempio:

```
public class A {  
    protected void m() { ... }  
    public static void main(String args[]) {  
        A a = new B();  
        a.m();  
    }  
}  
  
public class B extends A {  
    public void m() { ... }  
}
```


Overriding ed eccezioni

- Quando si esegue overriding di un metodo che dichiara di sollevare eccezioni C, il metodo ridefinito non può mai sollevare "più" tipi di eccezione (controllate) di quelli sollevati dall'originale. Può:
 - dichiarare a sua volta di sollevare eccezioni di classe C;
 - dichiarare di sollevare eccezioni di una sottoclasse di C;
 - dichiarare di non sollevare eccezioni.
- Non potrebbe, invece:
 - dichiarare di sollevare eccezioni di una superclasse di C o di una classe non legata a C da legami di ereditarietà.

Java Summary

- Objects
 - have fields and methods
 - alloc on heap, access by pointer, garbage collected
- Classes
 - Public, Private, Protected, Package (not exactly C++)
 - Can have static (class) members
 - Constructors and finalize methods
- Inheritance
 - Single inheritance
 - Final classes and methods

Java Summary (II)

- Subtyping
 - Determined from inheritance hierarchy
 - Class may implement multiple interfaces
- Virtual machine
 - Load bytecode for classes at run time
 - Verifier checks bytecode
 - Interpreter also makes run-time checks
 - type casts
 - array bounds
 - ...
 - Portability and security are main considerations

Some Highlights

- Dynamic lookup
 - Different bytecodes for by-class, by-interface
 - Search vtable + Bytecode rewriting or caching
- Subtyping
 - Interfaces instead of multiple inheritance
 - Awkward treatment of array subtyping (my opinion)
- Generics
 - Type checked, not instantiated, some limitations (`<T>...new T`)
- Bytecode-based JVM
 - Bytecode verifier
 - Security: security manager, stack inspection

Comparison with C++

- Almost everything is object + Simplicity - Efficiency
 - except for values from primitive types
- Type safe + Safety +/- Code complexity - Efficiency
 - Arrays are bounds checked
 - No pointer arithmetic, no unchecked type casts
 - Garbage collected
- Interpreted + Portability + Safety - Efficiency
 - Compiled to byte code: a generalized form of assembly language designed to interpret quickly.
 - Byte codes contain type information

Comparison

(cont'd)

- Objects accessed by ptr + Simplicity - Efficiency
 - No problems with direct manipulation of objects
- Garbage collection: + Safety + Simplicity - Efficiency
 - Needed to support type safety
- Built-in concurrency support + Portability
 - Used for concurrent garbage collection (avoid waiting?)
 - Concurrency control via synchronous methods
 - Part of network support: download data while executing
- Exceptions
 - As in C++, integral part of language design

Links

- **Enhancements in JDK 5**

- <http://java.sun.com/j2se/1.5.0/docs/guide/language/index.html>

- **J2SE 5.0 in a Nutshell**

- <http://java.sun.com/developer/technicalArticles/releases/j2se15/>

- **Generics**

- <http://www.langer.camelot.de/Resources/Links/JavaGenerics.htm>