

Java

Angelo Gargantini

Informatica III - 2015/16

M4 java su syllabus

Outline

1 .Language Overview

- History and design goals

2. Classes and Inheritance

- Object features
- Encapsulation
- Inheritance

3. Types and Subtyping

- Primitive and ref types
- Interfaces; arrays
- Exception hierarchy
- Subtype polymorphism and generic programming

- Saltiamo il resto

Origins of the language

- James Gosling and others at Sun, 1990 - 95
- Oak language for “set-top box”
 - small networked device with television display
 - graphics
 - execution of simple programs
 - communication between local program and remote site
 - no “expert programmer” to deal with crash, etc.
- Internet application
 - simple language for writing programs that can be transmitted over network

Design Goals

- Portability
 - Internet-wide distribution: PC, Unix, Mac
- Reliability
 - Avoid program crashes and error messages
- Safety
 - Programmer may be malicious
- Simplicity and familiarity
 - Appeal to average programmer; less complex than C++
- Efficiency
 - Important but secondary

General design decisions

- **Simplicity**
 - Almost everything is an object
 - All objects on heap, accessed through pointers
 - No functions, no multiple inheritance, no go to, no operator overloading, few automatic coercions
- **Portability and network transfer**
 - Bytecode interpreter on many platforms
- **Reliability and Safety**
 - Typed source and typed bytecode language
 - Run-time type and bounds checks
 - Garbage collection

Pro e contro di Java

	Portability	Safety	Simplicity	Efficiency
Interpreted	+	+		-
Type safe	+	+	+/-	+/-
Most values objects	+/-	+/-	+	-
Objects by means of pointers	+		+	-
Garbage collection	+	+	+	-
Concurrency support	+	+		

Java System

- The Java programming language
- Compiler and run-time system
 - Programmer compiles code
 - Compiled code transmitted on network
 - Receiver executes on interpreter (JVM)
 - Safety checks made before/during execution
- Library, including graphics, security, etc.
 - Large library made it easier for projects to adopt Java
 - Interoperability
 - Provision for “native” methods

Java Release History

- 1995 (1.0) – First public release
- 1997 (1.1) – Nested classes
 - Support for function objects
- 2001 (1.4) – Assertions
 - Verify programmers understanding of code
- 2004 (1.5) – Tiger
 - Generics, foreach, Autoboxing/Unboxing,
 - Typesafe Enums, Varargs, Static Import,
 - Annotations, concurrency utility library
- 2006 (1.6) – Mustang
- 2011 (1.7) – Dolphin
- 2014 (1.8)

Outline

- ➔ Objects in Java
 - Classes, encapsulation, inheritance
- ◆ Type system
 - Primitive types, interfaces, arrays, exceptions
- ◆ Generics (added in Java 1.5)
 - Basics, wildcards, ...

Language Terminology

- Class, object -
- Field –
- Method -
- Static members -
- this -
- Package - set of classes in shared namespace
- Native method -

Java Classes and Objects (2)

- Syntax similar to C++
- Object
 - has fields and methods
 - is allocated on heap, not run-time stack
 - accessible through reference (only ptr assignment)
 - garbage collected
- Dynamic lookup
 - Similar in behavior to other languages
 - Static typing => more efficient than Smalltalk
 - Dynamic linking, interfaces => slower than C++

Point Class

```
class Point {  
    static public Point O = new Point(0);  
    private int x;  
    Point(int xval) {x = xval;}    // constructor  
    protected void setX (int y) {x = y;}  
    public int  getX()    {return x;}  
}
```

- Visibility similar to C++, but not exactly (later slide)

Object initialization

- Java guarantees constructor call for each object
 - Memory allocated
 - Constructor called to initialize memory
 - Some interesting issues related to inheritance

We'll discuss later ...

- Cannot do this (would be bad C++ style anyway):
 - `Obj* obj = (Obj*)malloc(sizeof(Obj));`
- Static fields of class initialized at class load time
 - Talk about class loading later

Static fields and methods

- static field is one field for the entire class, instead of one per object.
- static method may be called without using an object of the class
 - static methods may be called before any objects of the class are created. Static methods can access only static fields and other static methods;
- Outside a class, a static member is usually accessed with the class name, as in `class_name.static_method(args)`,
-

```
class ... {
    /* static variable with initial value */
    static int x = initial_value;
    /* --- static initialization block --- */
    static {
        /* code to be executed once, when class is
        loaded */
    }
}
```

- the static initialization block of a class is executed once, when the class is loaded.

Garbage Collection and Finalize

- Objects are garbage collected
 - No explicit *free*
 - Avoids dangling pointers and resulting type errors
- Problem
 - What if object has opened file or holds lock?
- Solution
 - *finalize* method, **called by the garbage collector**
 - Before space is reclaimed, or when virtual machine exits
 - Space overflow is not really the right condition to trigger finalization when an object holds a lock...)
 - Important convention: call `super.finalize`
 - Don't design your Java programs such that correctness depends upon "timely" finalization.

Esercizio con gc e finalize

- Come usare finalize ...

Packages and visibility

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
No modifier (friendly)	Y	Y	N	N
private	Y	N	N	N

Estensione delle classi (3)

Inheritance

- Similar to Smalltalk, C++
- Subclass inherits from superclass
 - Single inheritance only (but Java has interfaces)
- Some additional features
 - Conventions regarding *super* in constructor and *finalize* methods
 - Final classes and methods

Example subclass

```
class ColorPoint extends Point {  
    // Additional fields and methods  
    private Color c;  
    protected void setC (Color d) {c = d;}  
    public Color  getC()   {return c;}  
    // Define constructor  
    ColorPoint(int xval, Color cval) {  
        super(xval); // call Point constructor  
        c = cval; } // initialize ColorPoint field  
}
```

Class *Object*

- Every class extends another class
 - Superclass is *Object* if no other class named
- Methods of class *Object*
 - `getClass` – return the `Class` object representing class of the object
 - `toString` – returns string representation of object
 - `equals` – default object equality (not ptr equality)
 - `hashCode`
 - `clone` – makes a duplicate of an object
 - `wait`, `notify`, `notifyAll` – used with concurrency
 - `finalize`

Constructors and Super

- Java guarantees constructor call for each object
- This must be preserved by inheritance
 - Subclass constructor must call super constructor
 - If first statement is not call to super, then call super() inserted automatically by compiler
 - If superclass does not have a constructor with no args, then this causes compiler error (yuck)
 - Exception to rule: if one constructor invokes another, then it is responsibility of second constructor to call super, e.g.,

```
ColorPoint() { ColorPoint(0,blue);}
```

is compiled without inserting call to super
- Different conventions for finalize and super
 - Compiler does not force call to super finalize

Final classes and methods

- Restrict inheritance
 - Final classes and methods cannot be redefined
- Example
 - `java.lang.String`
- Reasons for this feature
 - Important for security
 - Programmer controls behavior of all subclasses
 - Critical because subclasses produce subtypes
 - Compare to C++ virtual/non-virtual
 - Method is “virtual” until it becomes final



Altri argomenti

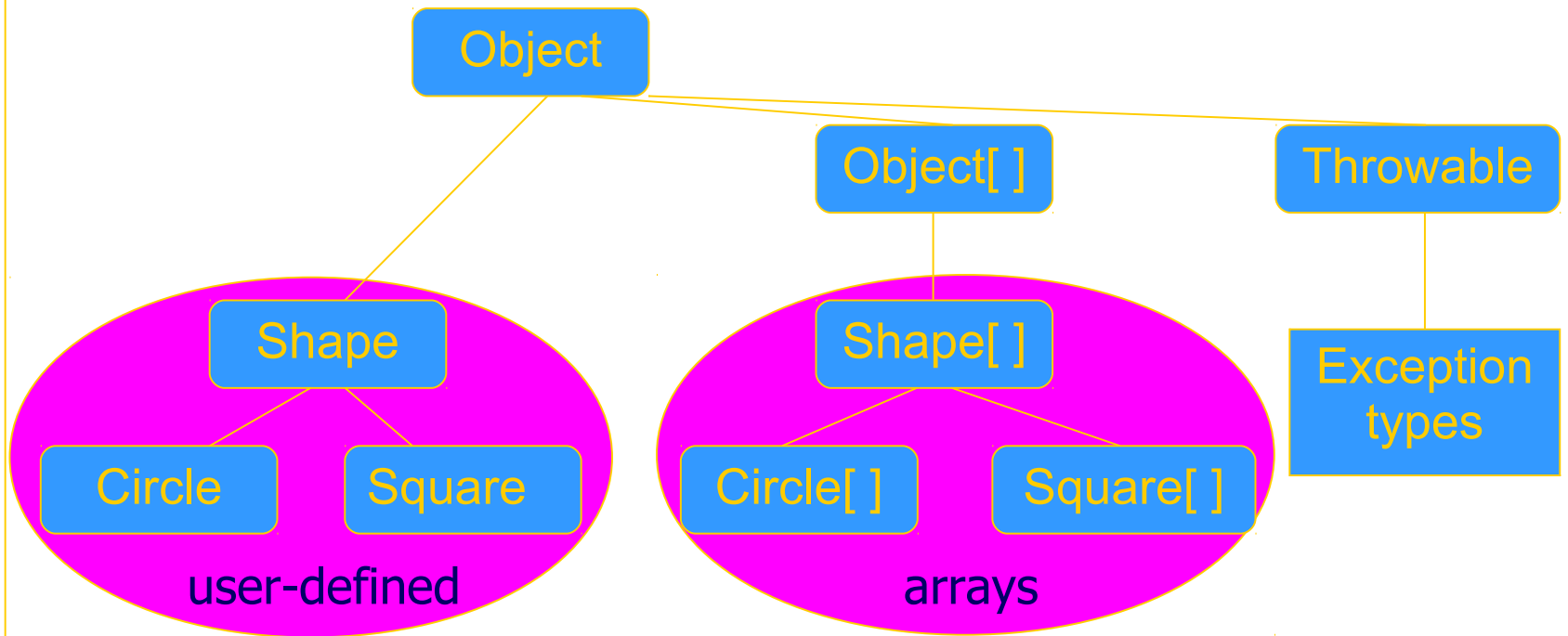
- Compatibilità di tipi e conversione
 - Sottoclassi e sottotipi
- Classi astratte e interfacce
- Ereditarietà e ridefinizione dei membri
- Binding dinamico

Java Types

- Two general kinds of times
 - Primitive types – *not* objects
 - Integers, Booleans, etc
 - Reference types
 - Classes, interfaces, arrays
 - No syntax distinguishing Object * from Object
- Static type checking
 - Every expression has type, determined from its parts
 - Some auto conversions, many casts are checked at run time
 - Example, assuming $A <: B$ (A sottotipo di B)
 - Can use A x and type
 - If B x, then can try to cast x to A
 - Downcast checked at run-time, may raise exception

Classification of Java types

Reference Types



Primitive Types



Subtyping

- Primitive types
 - Conversions: int -> long, double -> long, ...
- Class subtyping similar to C++
 - Subclass produces subtype
 - Single inheritance => subclasses form tree
- Interfaces
 - Completely abstract classes
 - no implementation
 - Multiple subtyping
 - Interface can have multiple subtypes (extends, implements)
- Arrays
 - Covariant subtyping – not consistent with semantic principles

Java class subtyping

- Signature Conformance
 - Subclass method signatures must conform to those of superclass
- Three ways signature could vary
 - Argument types
 - Return type
 - Exceptions

How much conformance is needed in principle?
- Java rule
 - Java 1.1: Arguments and returns must have identical types, may remove exceptions
 - Java 1.5: covariant return type specialization

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
 - ...
 -

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() {...}  
}
```

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

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
```

Interfacce (4)

Interface subtyping: example

```
interface Shape {
    public float center();
    public void rotate(float degrees);
}
interface Drawable {
    public void setColor(Color c);
    public void draw();
}
class Circle implements Shape, Drawable {
    // does not inherit any implementation
    // but must define Shape, Drawable methods
}
```

Properties of interfaces

- Flexibility
 - Allows subtype graph instead of tree
 - Avoids problems with multiple inheritance of implementations (remember C++ “diamond”)
- Cost
 - Offset in method lookup table not known at compile
 - Different bytecodes for method lookup
 - one when class is known
 - one when only interface is known
 - search for location of method
 - cache for use next time this call is made (from this line)

Tipi enumerativi (6)

Enumeration

- In prior releases, the standard way to represent an enumerated type was the int Enum pattern
- Not typesafe
- No namespace - You must prefix constants of an int enum with a string (in this case SEASON_)
- Printed values are uninformative

In Java5

```
public enum Season {  
    WINTER, SPRING, SUMMER, FALL }
```

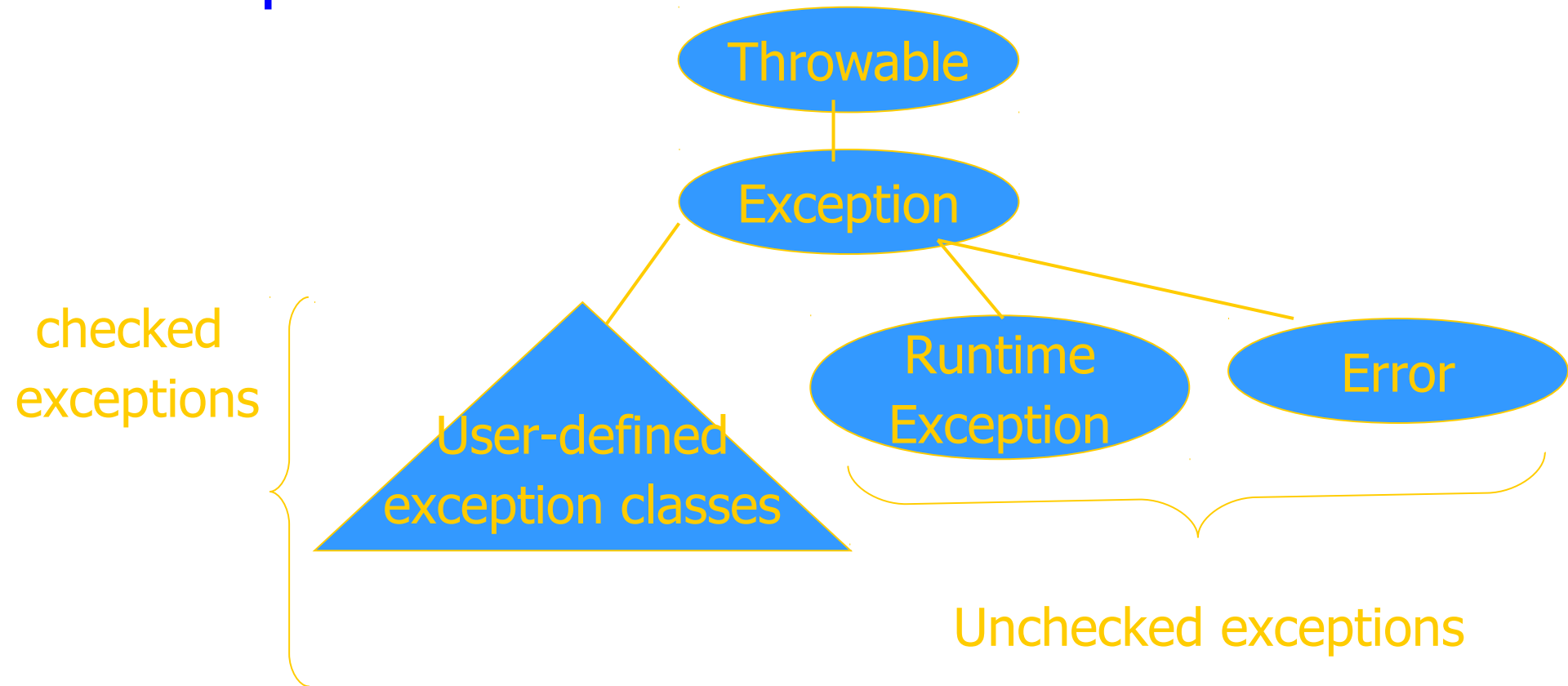
- Comparable
- toString which prints the name of the symbol
- static method that returns an array containing all of the values of the enum type in the order they are declared
 - for (Season s : Season.values()) ...

Eccezioni e asserzioni (12)

Java Exceptions

- Similar basic functionality to ML, C++
 - Constructs to *throw* and *catch* exceptions
 - Dynamic scoping of handler
- Some differences
 - An exception is an object from an exception class
 - Subtyping between exception classes
 - Use subtyping to match type of exception or pass it on ...
 - Similar functionality to ML pattern matching in handler
 - Type of method includes exceptions it can throw
 - Actually, only subclasses of Exception (see next slide)

Exception Classes



- If a method may throw a checked exception, then this must be in the type of the method

Try/finally blocks

- Exceptions are caught in try blocks

```
try {  
    statements  
} catch (ex-type1 identifier1) {  
    statements  
} catch (ex-type2 identifier2) {  
    statements  
} finally {  
    statements  
}
```

- Implementation: finally compiled to jsr

Why define new exception types?

- Exception may contain data
 - Class Throwable includes a string field so that cause of exception can be described
 - Pass other data by declaring additional fields or methods
- Subtype hierarchy used to catch exceptions

```
catch <exception-type> <identifier> { ... }
```

will catch any exception from any subtype of exception-type and bind object to identifier

Binding Dinamico in Java

Overload vs Override

- Overload = più metodi o costruttori con lo stesso nome ma diversa segnatura
 - Segnatura: nome del metodo e lista dei tipi dei suoi argomenti
- L'overloading viene risolto in fase di compilazione
- Esempio

```
public static double valoreAssoluto(double x) {  
    if (x > 0) return x;  
    else return -x;  
}
```

```
public static int valoreAssoluto(int x) {  
    return (int) valoreAssoluto((double) x);  
}
```

Compilazione: scelta segnatura

- In compilazione viene **scelta la segnatura del metodo da eseguire** in base:
 - (1) al **tipo del riferimento** utilizzato per invocare il metodo
 - (2) al **tipo degli argomenti** indicati nella chiamata

Esempio

- A r;...
- r.m(2)
- Il compilatore cerca fra tutte le segnature di metodi di nome **m** disponibili per il tipo **A** quella **"più adatta"** per gli argomenti specificati

Esempio

A r;

...

r.m(2)

- Se le signature disponibili per il tipo **A** sono:

int m(byte b)

int m(long l)

int m(double d)

- il compilatore sceglie la seconda

Overriding

- Quando si riscrive in una sottoclasse un metodo della superclasse con la **stessa segnatura**.
- L'overriding viene risolto **in fase di esecuzione**
- **Compilazione:**
- scelta della segnatura: il compilatore stabilisce **la segnatura** del metodo da eseguire (early binding)
- **Esecuzione:**
- scelta del metodo: Il metodo da eseguire, tra quelli con la segnatura selezionata, viene scelto al momento dell'esecuzione, sulla base del **tipo dell'oggetto** (late binding)

Fase di compilazione

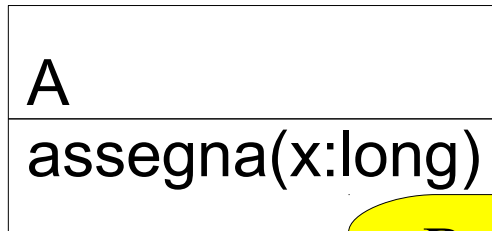
(1) Scelta delle signature "candidate"

- Il compilatore individua le signature che possono **soddisfare la chiamata**
 - (a) **compatibile con gli argomenti utilizzati nella chiamata**
il numero dei parametri nella signature `e uguale al numero degli argomenti utilizzati ogni argomento `e di un tipo assegnabile al corrispondente parametro
 - (b) **accessibile al codice chiamante**
- Se non esistono signature candidate, il compilatore segnala un errore.

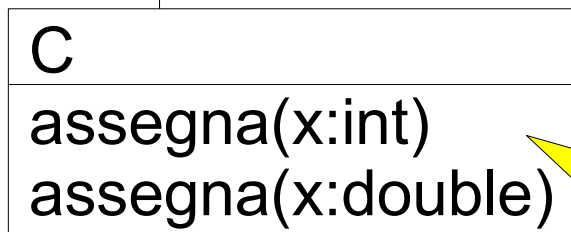
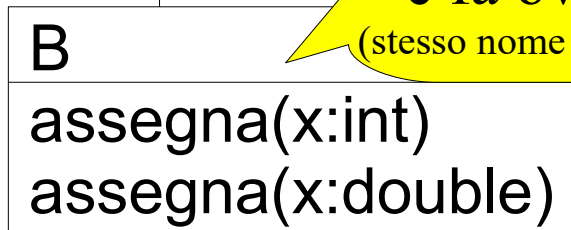
(2) Scelta della signature "più specifica"

- Tra le signature candidate, il compilatore seleziona quella che richiede il minor numero di promozioni

Esempio 1



B eredita da A
e fa overloading
(stesso nome segnatura diversa)



C eredita da B
e fa overriding
(stesso nome e segnatura)

A alfa;

- alfa.assegna(2)

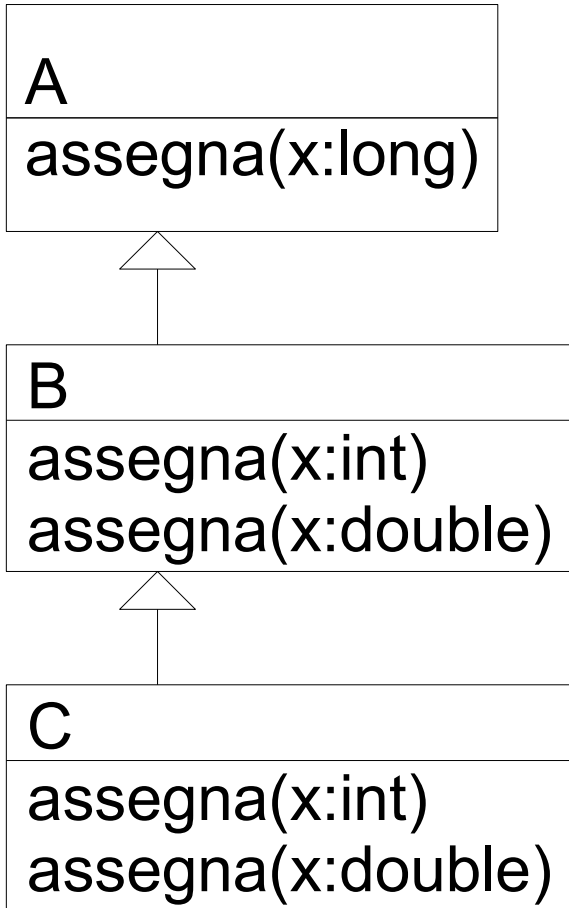
Una segnatura candidata:

assegna(long x)

- alfa.assegna(2.0)

Nessuna segnatura
candidata (errore)

Esempio 2



B beta;

`beta.assegna(2)`

Tre segnature candidate:

- `assegna(int x)`
- `assegna(double x)`
- `assegna(long x)`
- La più specifica è `assegna(int x)`

Ambiguità

- Se per l'invocazione:
- `z(1, 2)`
- le signature candidate sono:
- `z(double x, int y)`
- `z(int x, double y)`
- Il compilatore non `e in grado di individuare la segnatura pi`u specifica e segnala un messaggio di errore

Esecuzione: scelta del metodo

- La JVM sceglie il metodo da eseguire **sulla base della classe dell'oggetto** usato nell'invocazione
 - cerca un metodo con la segnatura selezionata in fase di compilazione
 - risalendo la gerarchia delle classi a partire dalla classe dell'oggetto che deve eseguire il metodo

Esempio 1

A alpha = new B();

alpha.assegna(2l)

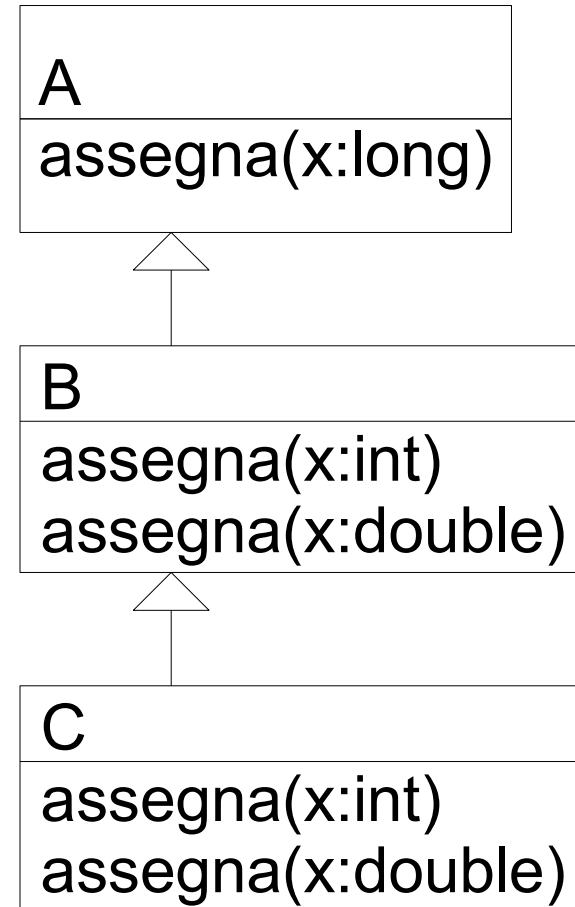
EB: segnatura selezionata in

A: **assegna(long x)**

LB: Ricerca a partire da B un
metodo **assegna(long)**

Esegue il metodo di A

In questo caso metodo
selezionato in EB ed
eseguito coincidono



Esempio 2

B beta = new C()

beta.assegna(2)

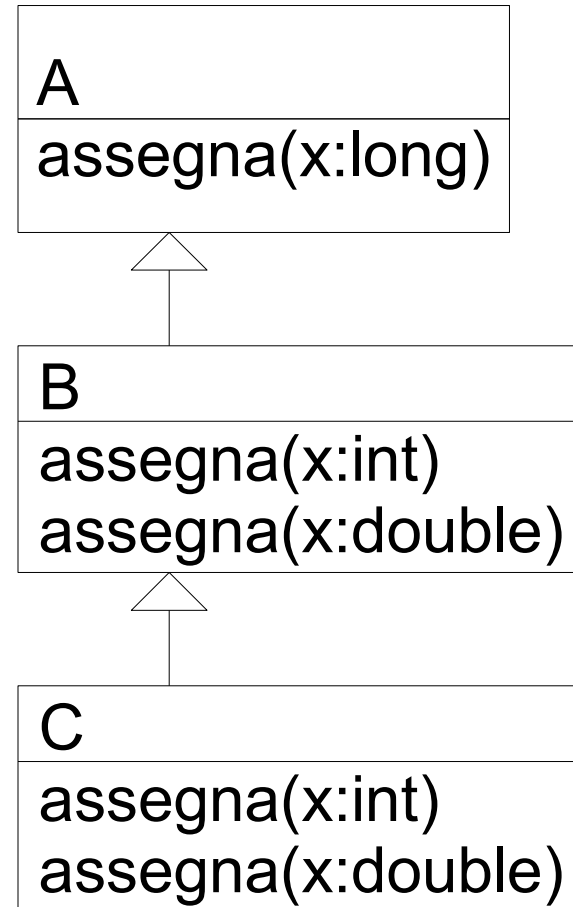
EB: segnatura selezionata di

B: **assegna(int x)**

LB: Ricerca a partire da C un
metodo **assegna(int)**

Esegue il metodo di C

Come volevo,
poichè ho ridefinito il metodo



Esempio 3

A alfa = new C()

alfa.assegna(2)

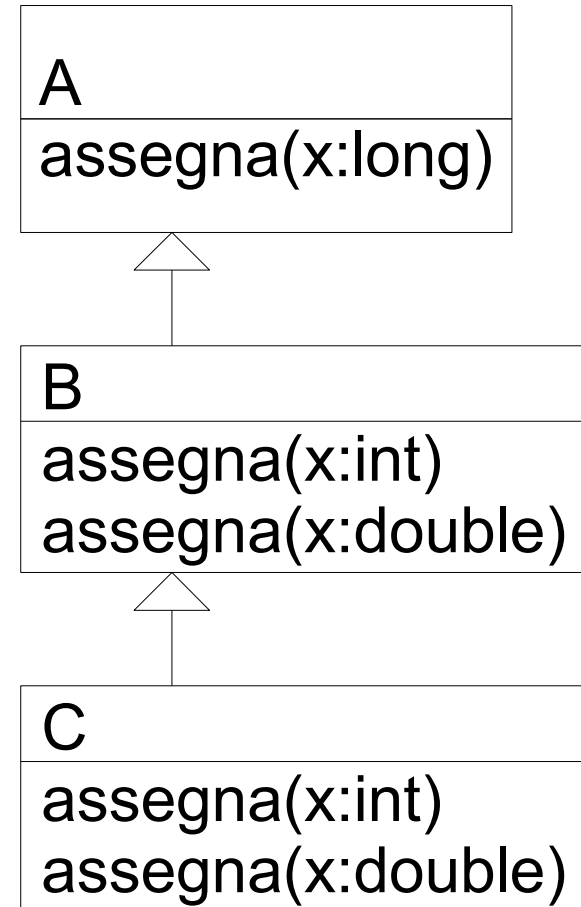
EB Una segnatura candidata:

assegna(long x)

LB: Ricerca a partire da C un
metodo assegna(long)

Esegue il metodo di A anche
se 2 è int !!!

E' dovuto al fatto che non ho
ridefinito il metodo di A



Attenzione

- Quando si ridefiniscono i metodi in java bisogna usare la stessa segnatura !!
- Vedi il problema con equals

```
class A {  
    int x;  
    A(int y){x = y;}  
    public equals(A a){ return (x == a.x);}  
}
```

```
Object a1 = new A(3);
```

```
A a2 = new A(3);
```

```
a1.equals(a2);
```

Outline

- Objects in Java
 - Classes, encapsulation, inheritance
- Type system
 - Primitive types, interfaces, arrays, exceptions
- Generics (added in Java 1.5)
 - Basics, wildcards, ...
- ◆ Virtual machine
 - ◆
 - Loader, verifier, linker, interpreter
 - Bytecodes for method lookup
- ◆ Security issues

Enhancements in JDK 5 (= Java 1.5)

- Enhanced for Loop
 - for iterating over collections and arrays
- Autoboxing/Unboxing
 - automatic conversion between primitive, wrapper types
- Typesafe Enums
 - enumerated types with arbitrary methods and fields
- Varargs
 - puts argument lists into an array; variable-length argument lists
- Static Import
 - avoid qualifying static members with class names
- Annotations (Metadata)
 - enables tools to generate code from annotations (JSR 175)
- Generics
 - polymorphism and compile-time type safety

varargs

- Varargs sono usati per dichiarare un metodo che possa prendere in ingresso un oggetto, n- oggetti o un array di oggetti.
- Esempio
- `print(String ... s)`
- Permette le seguenti chiamate:
- `print("pippo")`
- `print("pippo", "pluto")`
- `print(new String[]{"a", "b", "c"})`
- Il tipo del parametro formale di un varargs è un array

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 $S <: T$
- 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 degli 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

- Posso specializzare mediante ereditarietà:

```
IntegerList extends ArrayList{  
    @Override  
    add(Object o){  
        // check o is Integer ...  
    }  
}
```

Generics

Invece mediante i generici:

```
class Stack<A> {  
    void push(A a) { ... }  
    A pop() { ... }
```

Annoto con <> il TIPO
Generico. A non è una classe

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

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; }  
  
}
```

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:

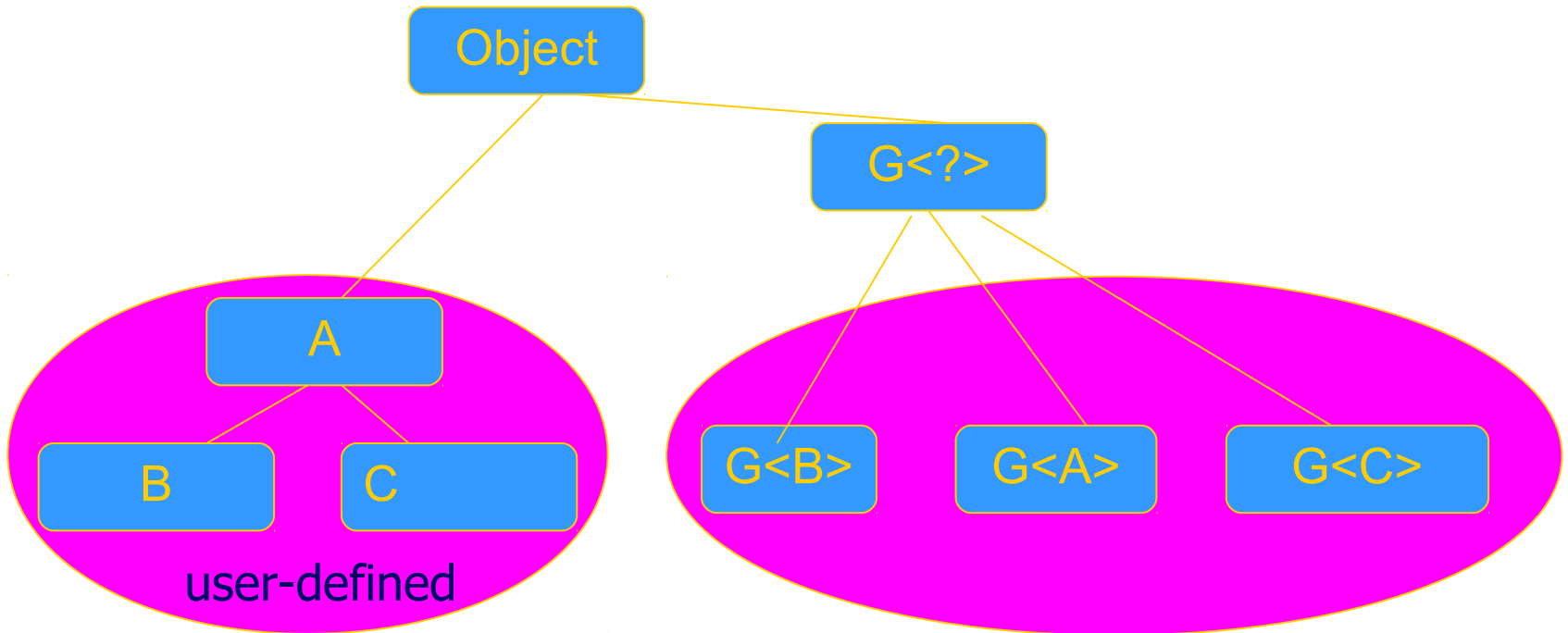
```
void printCollection(Collection c) {...}
```

- Con i generici???

```
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);  
    }  
}
```

No covarianza dei generics → conseguenze

- Nota che se $S <: T$ una classe $P < S >$ non è sottotipo di $P < T >$
 - In questo modo non ho i problemi degli array
 - Esempio: `Studente <: Persona`, non ho che `List < Studente > <: List < Persona >`
- Se un metodo chiede $P < T >$ non posso passare $P < T >$
 - Esempio:
 - `stampaAnagrafica(List < Persona > p)`
 - `List < Studente > ls;`
 - `stampaAnagrafica(ls)` non compila
 - Posso usare i wildcards per rendere il metodo più tollerante
 - `stampaAnagrafica(List < ? extends Persona > p)`

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

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

This expression must typecheck
Use "Key extends Hashable"

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

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() { ... }  
    ...  
}
```

Another example ...

```
interface LessAndEqual<I> {
    boolean lessThan(I);
    boolean equal(I);
}
class Relations<C extends LessAndEqual<C>> extends C {
    boolean greaterThan(Relations<C> a) {
        return a.lessThan(this);
    }
    boolean greaterEqual(Relations<C> a) {
        return greaterThan(a) || equal(a);
    }
    boolean notEqual(Relations<C> a) { ... }
    boolean lessEqual(Relations<C> a) { ... }
    ...
}
```


Wildcard e generics

- Alcune volte non si vuole specificare esattamente il tipo ma si vuole essere più permissivi
-
- Studente 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>`

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.
- 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>)`.

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

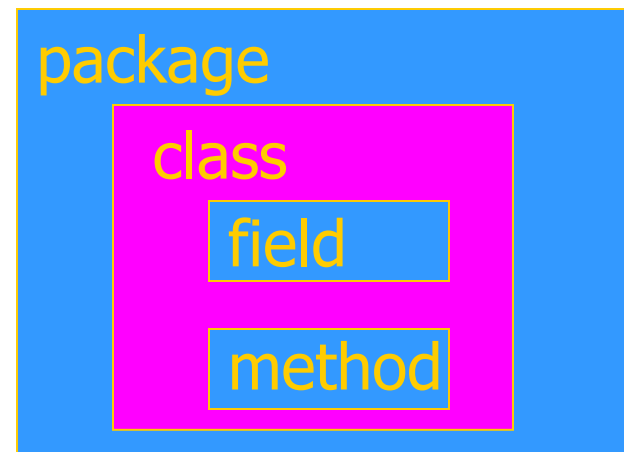
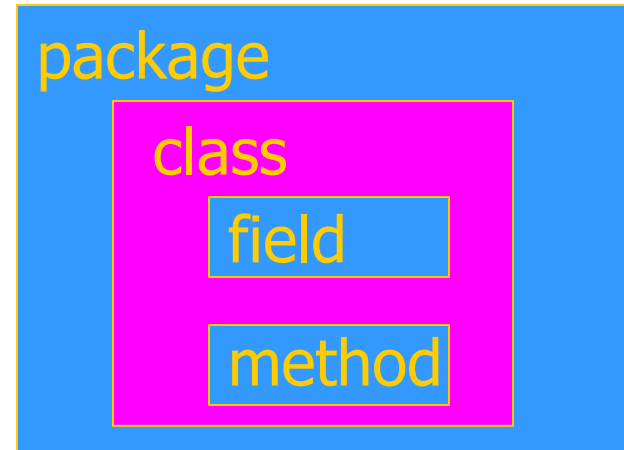
```
Stack<Integer> st =  
    new Stack<Integer>();  
st.push(12);  
...  
int i = st.pop();
```

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()`



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>