Java

Angelo Gargantini

Informatica III /2007/08

Outline

- Language Overview
  - History and design goals
- Classes and Inheritance
  - Object features
  - Encapsulation
  - Inheritance
- Types and Subtyping
  - Primitive and ref types
  - Interfaces; arrays
  - Exception hierarchy
  - Subtype polymorphism and generic programming
- Virtual machine overview
  - Loader and initialization
  - Linker and verifier
  - Bytecode interpreter
- Method lookup
  - four different bytecodes
- Verifier analysis
- Implementation of generics
- Security
  - Buffer overflow
  - Java “sandbox”
  - Type safety and attacks

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

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

Improvements through Java Community Process

Enhancements in JDK 5 (= Java 1.5)

- Generics
  - polymorphism and compile-time type safety
- 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)
- Concurrency utility library,

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

Language Terminology

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

Java Classes and Objects

- 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;
    protected void setX (int y)  {x = y;}
    public int  getX()     {return x;}
    Point(int xval) {x = xval;}       // constructor
}
```

– 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

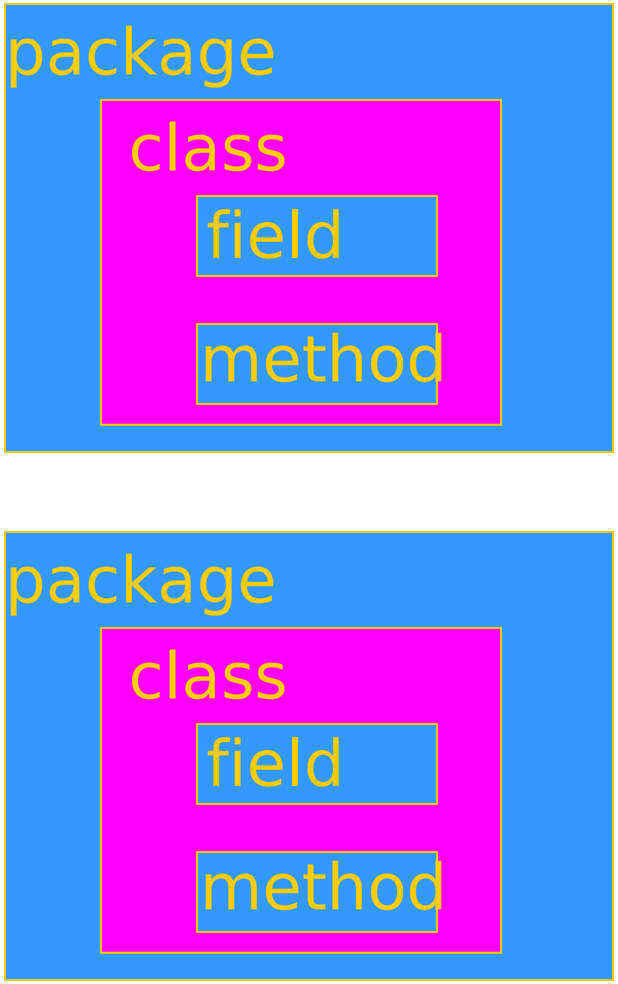
    We'll discuss later …
- Cannot do this (would be bad C++ style anyway):
  - Obj* obj = (Obj*)malloc(sizeof(Obj));
- Static fields of class initialized at class load time
  - Talk about class loading later

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

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

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

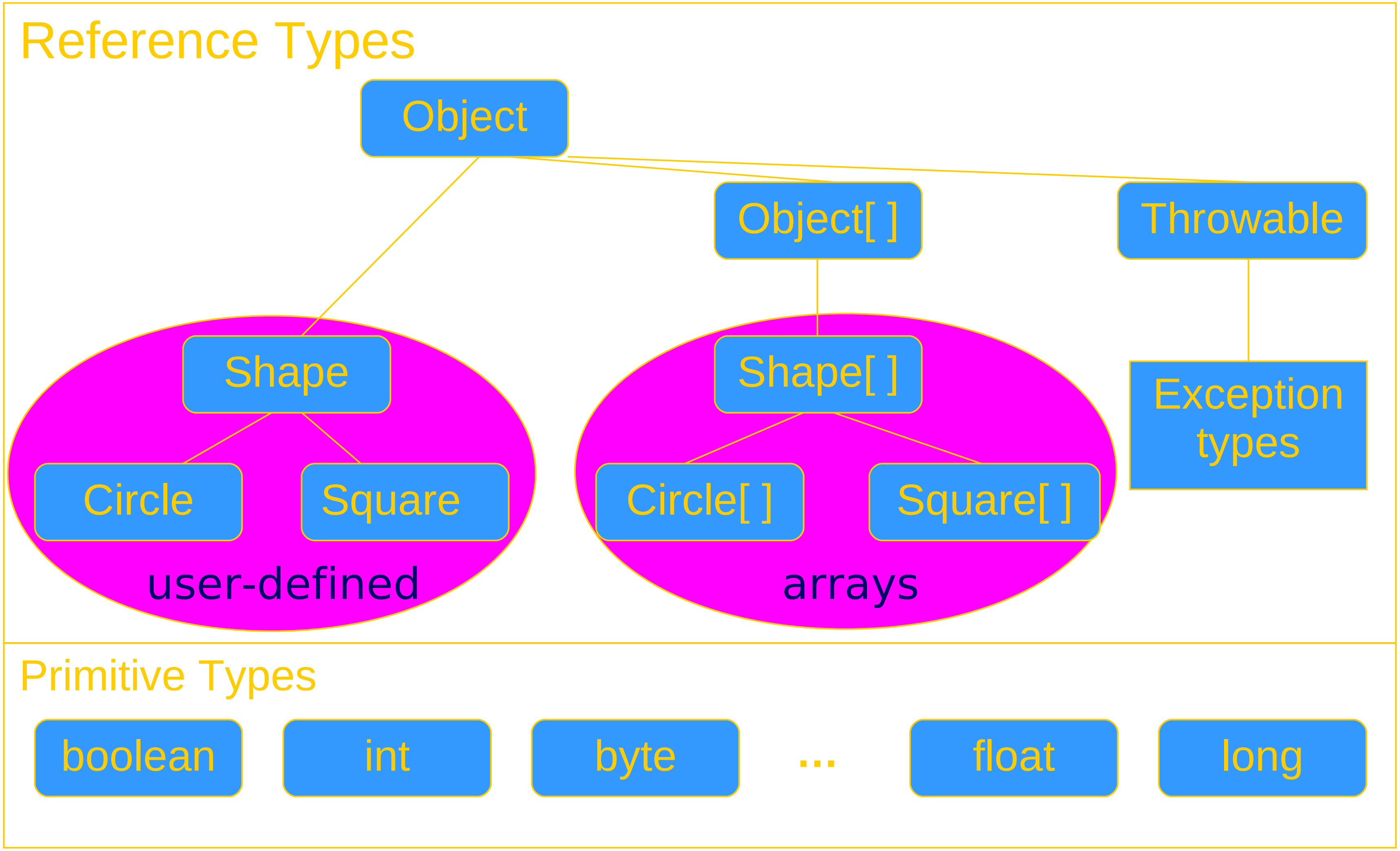
Outline

- 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

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



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 means that arguments, return values, or exceptions of overriding methods can be of subtypes of the original types.
- In Java 5 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() {...}
}
```

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)

Overload vs Override

- Overlod = 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
-

```
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 segnature 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

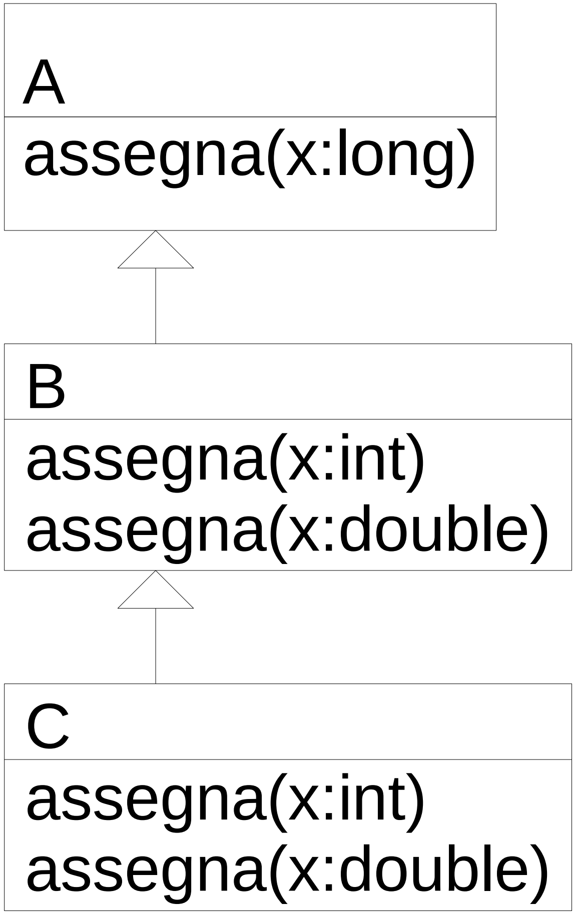
(1) Scelta delle segnature “candidate”

- Il compilatore individua le segnature che possono soddisfare la chiamata
  - (a) compatibile con gli argomenti utilizzati nella chiamata
    il numero dei parametri nella segnatura `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 segnature candidate, il compilatore segnala un errore.

(2) Scelta della segnatura “pi`u specifica”

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

Esempio 1



A alfa;

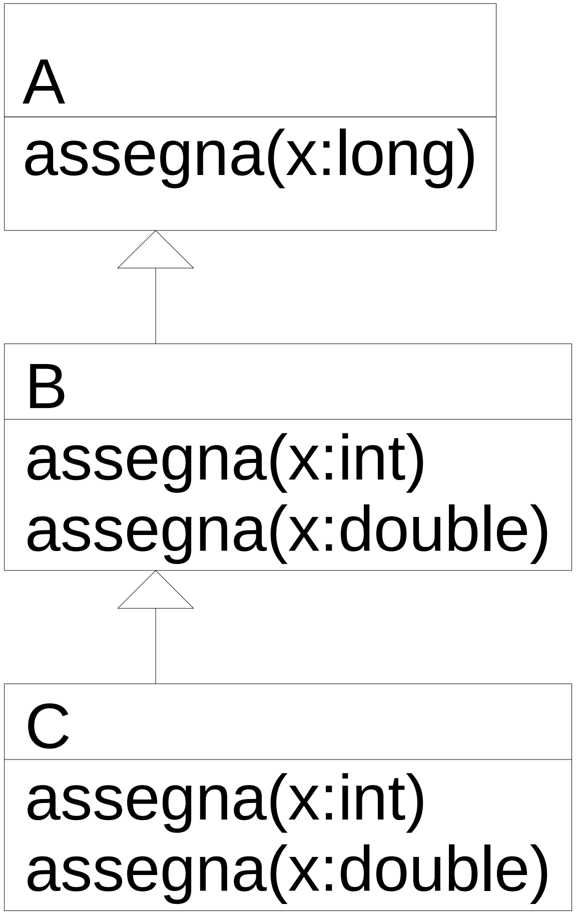
- alfa.assegna(2)

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 pi`u specifica `e assegna(int x)

Ambiguità

- Se per l’invocazione:
- z(1, 2)
- le segnature 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 del tipo 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

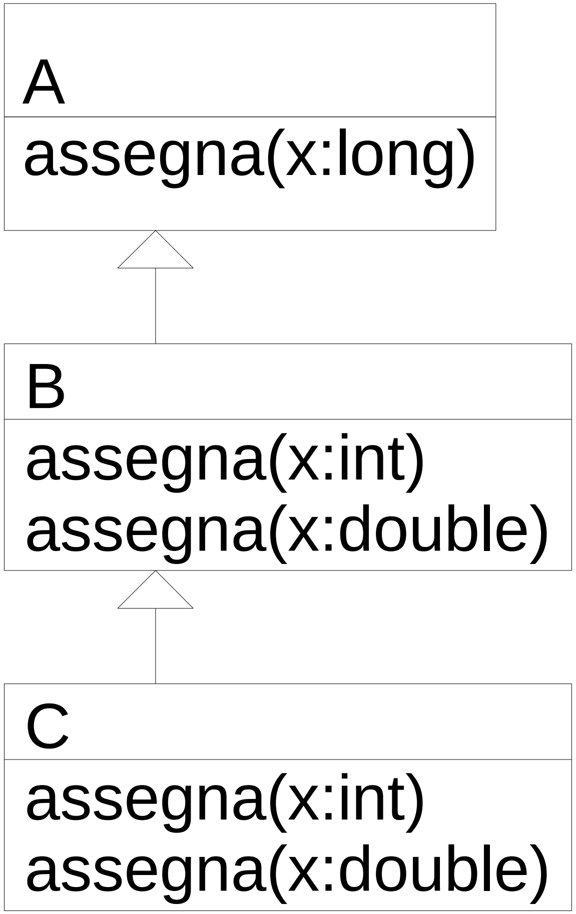
A alfa = new C()

alfa.assegna(2)

Una segnatura candidata:
  assegna(long x)

Ricerca a partire da C un metodo assegna(long)

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



Esempio 2

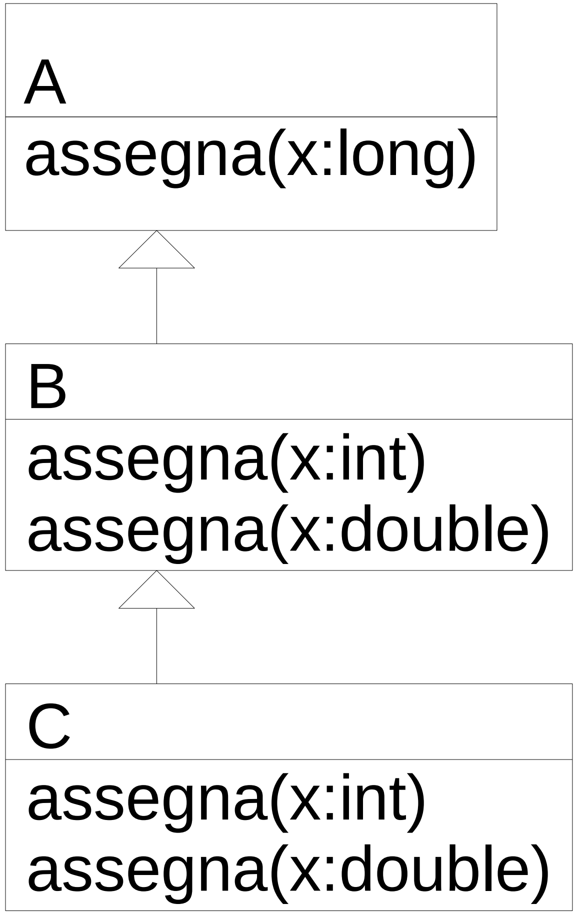
B beta = new C()

beta.assegna(2)

segnatura selezionata:
 assegna(int x)

Ricerca a partire da C un metodo assegna(int)

Esegue il metodo di C



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);
```

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
```

Covariance problem again …

- Remember Simula problem
  - If A <: B, then A ref <: B ref
  - Needed run-time test to prevent bad assignment
  - Covariance for assignable cells is not right in principle
- Explanation
  - interface of "T reference cell" is

    put : $T \rightarrow T$ ref

    get : T ref $\rightarrow T$
  - Remember covariance/contravariance of functions

Afterthought on Java arrays

Date: Fri, 09 Oct 1998 09:41:05 -0600

From: bill joy

Subject: …[discussion about java genericity]

actually, java array covariance was done for less noble reasons …: it made some generic "bcopy" (memory copy) and like operations much easier to write...

I proposed to take this out in 95, but it was too late (...).

i think it is unfortunate that it wasn't taken out...

it would have made adding genericity later much cleaner, and [array covariance] doesn't pay for its complexity today.

wnj

But compare this to C++!!

- Access by pointer: you can't do array subtyping.

```
B* barr[15];
A* aarr[] = barr;   // not allowed
```

- Direct naming: allowed, but you get garbage !!

```
B barr[15];
A aarr[] = barr;
```

aarr[k] translates to *(aarr+sizeof(A)*k)

barr[k] translates to *(barr+sizeof(B)*k)

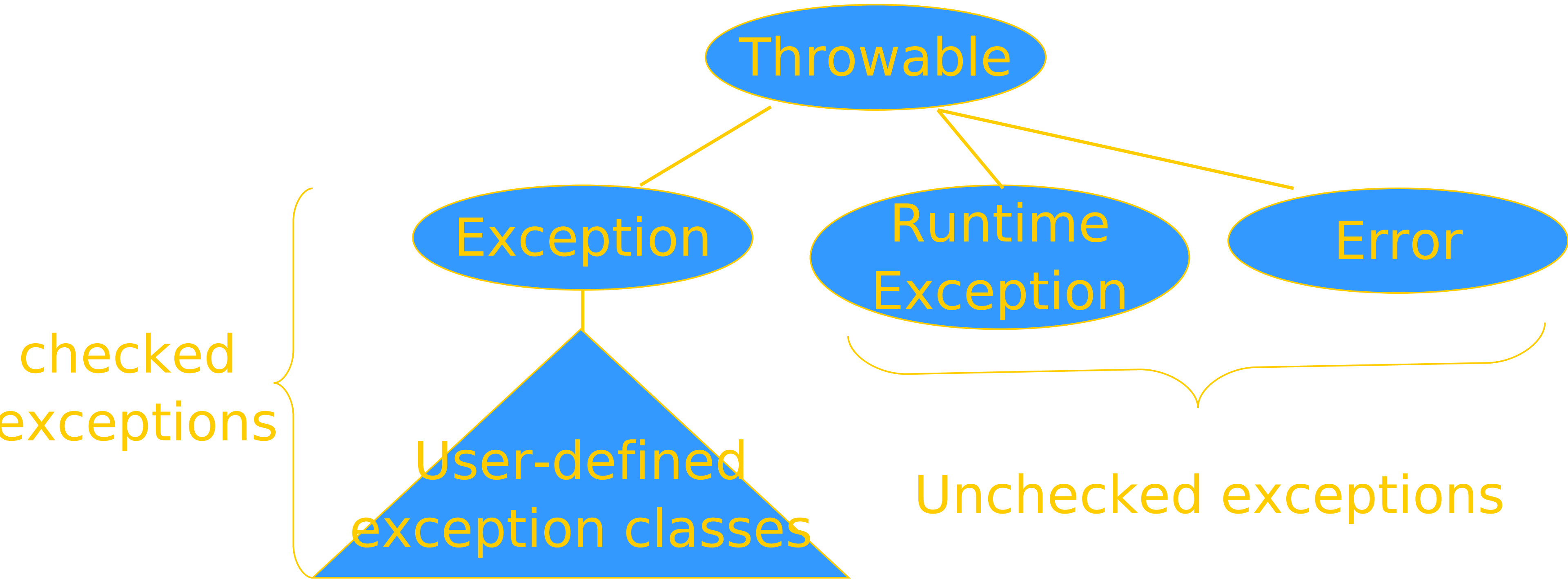
If sizeof(B) != sizeof(A), you just grab random bits.

Is there any sense to this?

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

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

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

Example generic construct: Stack

- Stacks possible for any type of object
  - For any type t, can have type stack_of_t
  - Operations push, pop work for any type
- In C++, would write generic stack class

```
template <type t> class Stack {
          private: t data;  Stack<t> * next;
          public: void    push (t* x) { … }
                    t*  pop   (     ) { … }
};
```

- What can we do in Java?

Java 1.0 vs Generics

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

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

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

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

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

Java Generics (Java 1.5, “Tiger”)

- Adopts syntax on previous slide
- 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();
```

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 …>

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

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

```
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>> {
        T queue[ ] ;   …
        void insert(T t) {
              … if ( t.compareTo(queue[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) { ... }
        ...
}
```

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

- Dichiara una classe A che ha come membro un intero
- Dichiara un classe B extends A che ha un metodo equals(B a)
- Dichiara una classe C extends A che ha un metodo equals(Object)
- Implementa i metodi toString in modo che stampino “A”, “B” e “C” e il valore dell'intero
- Dichiara 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???

Enumeration

- In prior releases, the standard way to represent an enumerated type was the int Enum pattern

```
// int Enum Pattern - has severe problems!
public static final int SEASON_WINTER = 0;
public static final int SEASON_SPRING = 1;
public static final int SEASON_SUMMER = 2;
public static final int SEASON_FALL   = 3;
```

- 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 `values` 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()) ...

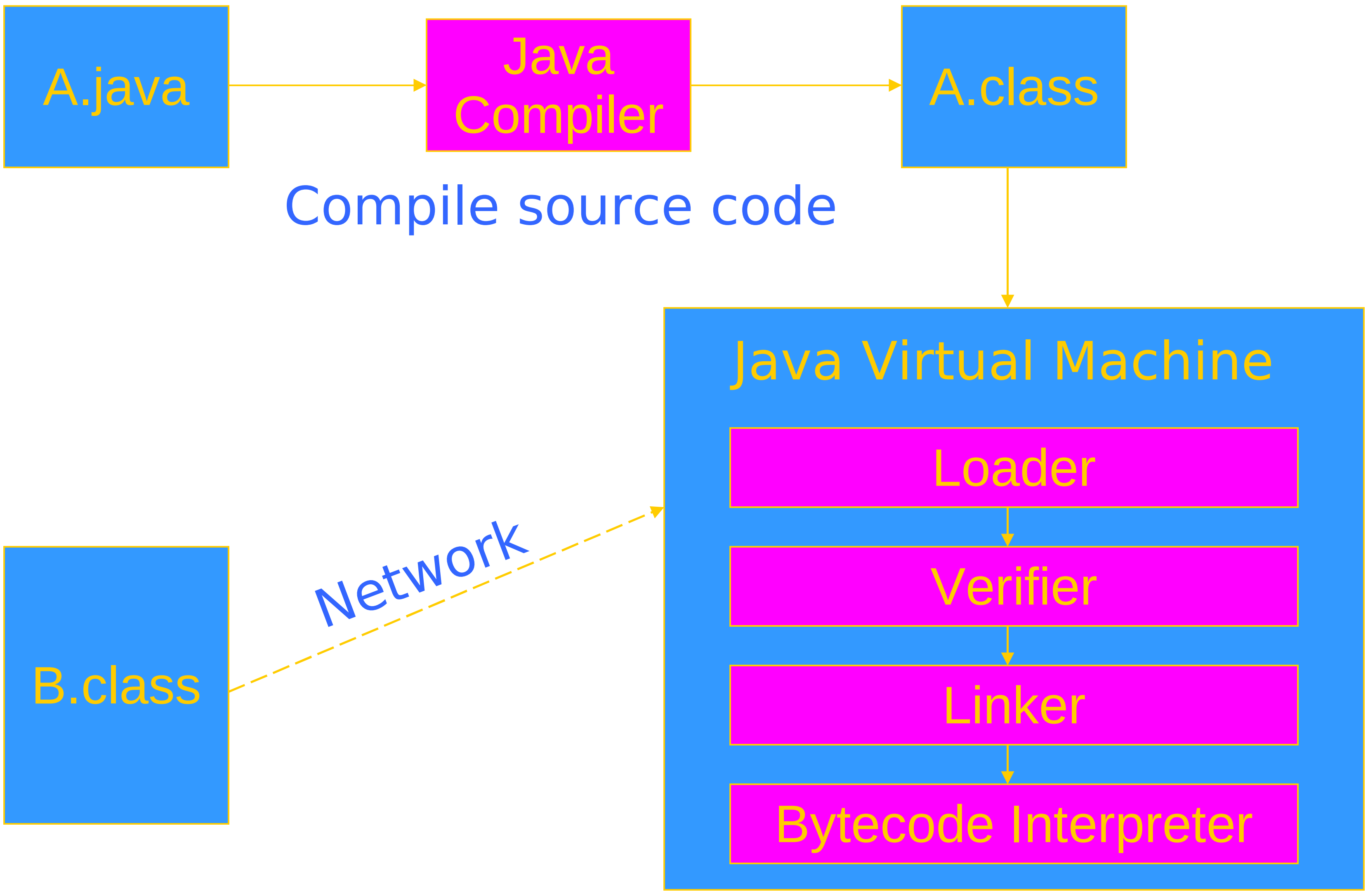
Outline

- 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
  - Bytecode verifier (example: initialize before use)
  - Implementation of generics
- Security issues

Java Implementation

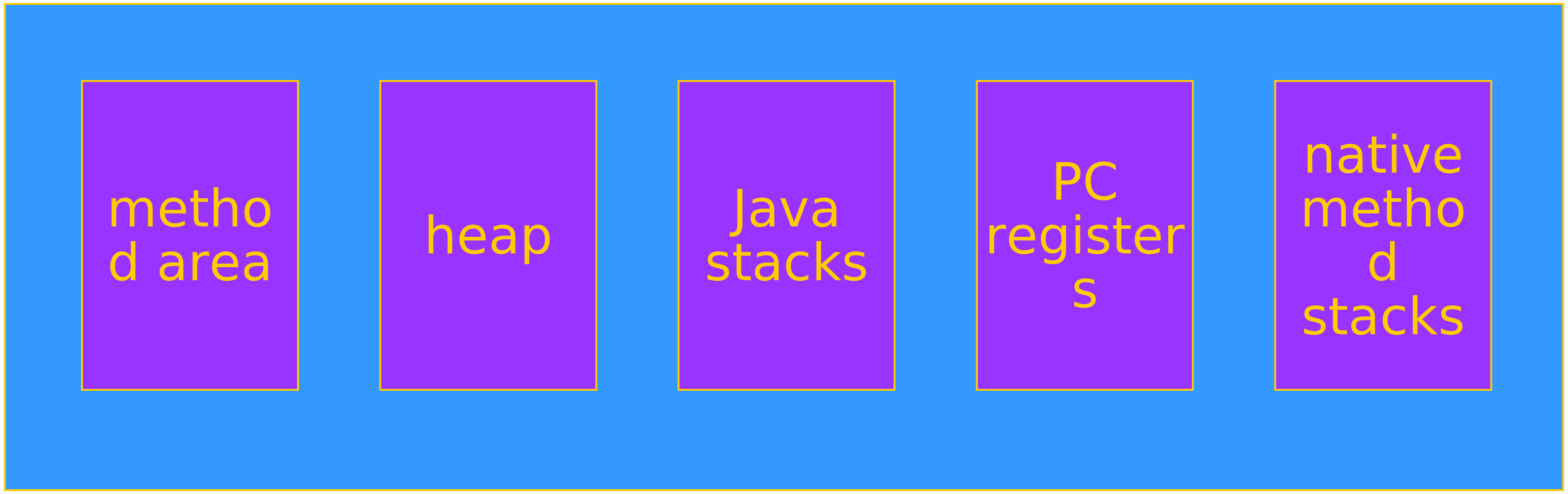
- Compiler and Virtual Machine
  - Compiler produces bytecode
  - Virtual machine loads classes on demand, verifies bytecode properties, interprets bytecode
- Why this design?
  - Bytecode interpreter/compilers used before
    - Pascal “pcode”; Smalltalk compilers use bytecode
  - Minimize machine-dependent part of implementation
    - Do optimization on bytecode when possible
    - Keep bytecode interpreter simple
  - For Java, this gives portability
    - Transmit bytecode across network

Java Virtual Machine Architecture



JVM memory areas

- Java program has one or more threads
- Each thread has its own stack
- All threads share same heap



Class loader

- Runtime system loads classes as needed
  - When class is referenced, loader searches for file of compiled bytecode instructions
- Default loading mechanism can be replaced
  - Define alternate ClassLoader object
    - Extend the abstract ClassLoader class and implementation
    - ClassLoader does not implement abstract method loadClass, but has methods that can be used to implement loadClass
  - Can obtain bytecodes from alternate source
    - VM restricts applet communication to site that supplied applet

Example issue in class loading and linking:

Static members and initialization

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

- Initialization is important
  - Cannot initialize class fields until loaded
- Static block cannot raise an exception
  - Handler may not be installed at class loading time

JVM Linker and Verifier

- Linker
  - Adds compiled class or interface to runtime system
  - Creates static fields and initializes them
  - Resolves names
    - Checks symbolic names and replaces with direct references
- Verifier
  - Check bytecode of a class or interface before loaded
  - Throw VerifyError exception if error occurs

Verifier

- Bytecode may not come from standard compiler
  - Evil hacker may write dangerous bytecode
- Verifier checks correctness of bytecode
  - Every instruction must have a valid operation code
  - Every branch instruction must branch to the start of some other instruction, not middle of instruction
  - Every method must have a structurally correct signature
  - Every instruction obeys the Java type discipline

Last condition is fairly complicated .

Bytecode interpreter

- Standard virtual machine interprets instructions
  - Perform run-time checks such as array bounds
  - Possible to compile bytecode class file to native code
- Java programs can call native methods
  - Typically functions written in C
- Multiple bytecodes for method lookup
  - invokevirtual - when class of object known
  - invokeinterface - when interface of object known
  - invokestatic - static methods
  - invokespecial - some special cases

Type Safety of JVM

- Run-time type checking
  - All casts are checked to make sure type safe
  - All array references are checked to make sure the array index is within the array bounds
  - References are tested to make sure they are not null before they are dereferenced.
- Additional features
  - Automatic garbage collection
  - No pointer arithmetic

    If program accesses memory, that memory is allocated to the program and declared with correct type

JVM uses stack machine

- Java

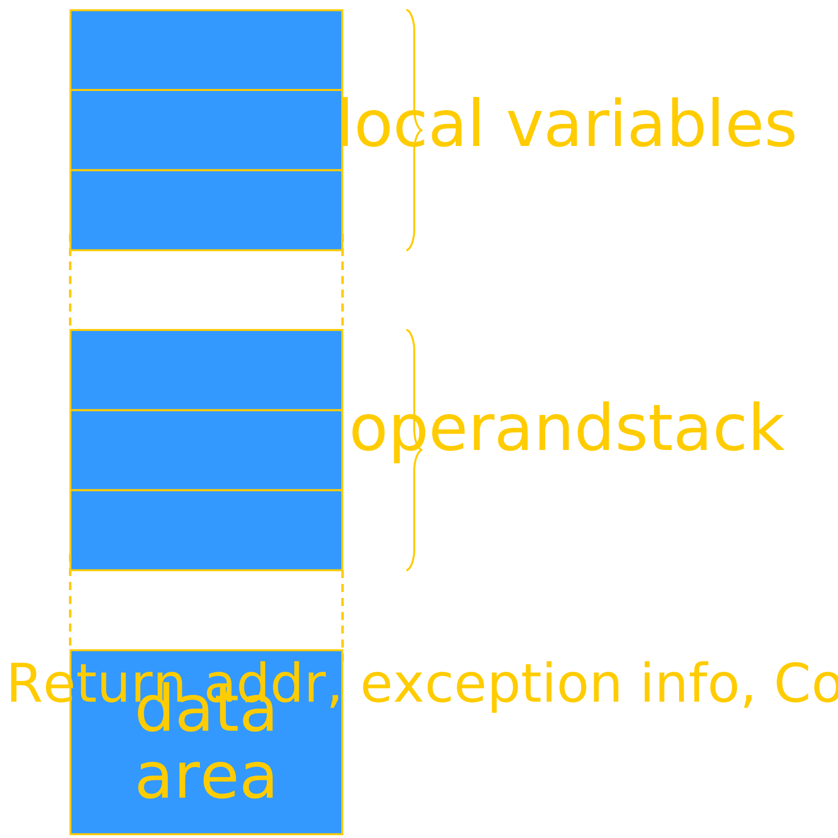
```
Class A extends Object {
    int i
    void f(int val) { i = val + 1;}
}
```

- Bytecode

```
Method void f(int)
    aload 0        ; object ref this
    iload 1        ; int val
    iconst 1
    iadd           ; add val +1
    putfield #4 <Field int i>
    return
```

refers to const pool

JVM Activation Record



Field and method access

- Instruction includes index into constant pool
  - Constant pool stores symbolic names
  - Store once, instead of each instruction, to save space
- First execution
  - Use symbolic name to find field or method
- Second execution
  - Use modified “quick” instruction to simplify search

invokeinterface <method-spec>

- Sample code

  void add2(Incrementable x) { x.inc(); x.inc(); }

- Search for method
  - find class of the object operand (operand on stack)
    - must implement the interface named in <method-spec>
  - search the method table for this class
  - find method with the given name and signature
- Call the method
  - Usual function call with new activation record, etc.

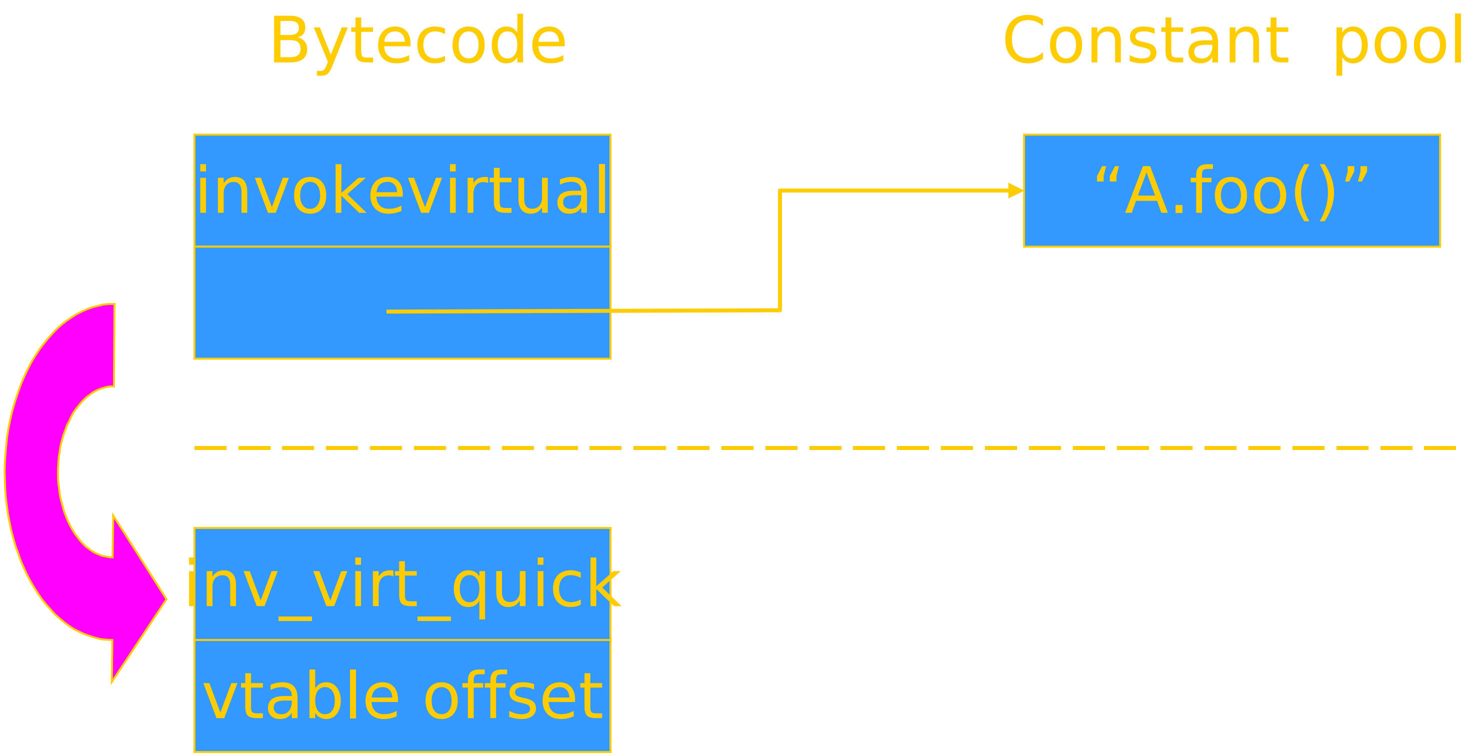
Why is search necessary?

```
interface Incrementable {
    public void inc();
}
class IntCounter implements Incrementable {
    public void add(int);
    public void inc();
   public int value();
}
class FloatCounter implements Incrementable {
    public void inc();
    public void add(float);
    public float value();
}
```

invokevirtual <method-spec>

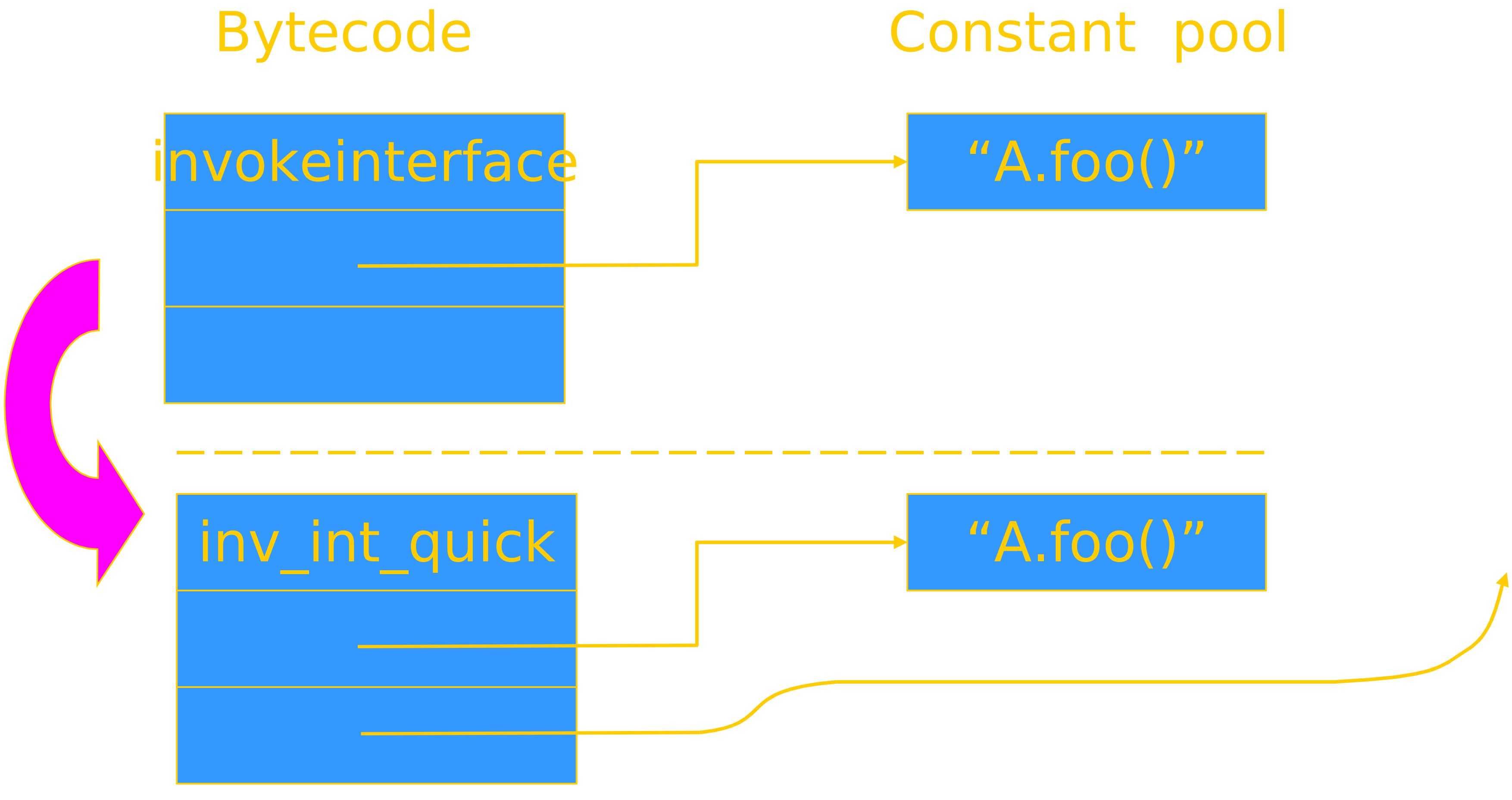
- Similar to invokeinterface, but class is known
- Search for method
  - search the method table of this class
  - find method with the given name and signature
- Can we use static type for efficiency?
  - Each execution of an instruction will be to object from subclass of statically-known class
  - Constant offset into vtable
    - like C++, but dynamic linking makes search useful first time
  - See next slide

Bytecode rewriting: invokevirtual



- After search, rewrite bytcode to use fixed offset into the vtable. No search on second execution.

Bytecode rewriting: invokeinterface



Cache address of method; check class on second use

Bytecode Verifier

- Let's look at one example to see how this works
- Correctness condition
  - No operations should be invoked on an object
  - until it has been initialized
- Bytecode instructions
  - new ⟨class⟩   allocate memory for object
  - init ⟨class⟩   initialize object on top of stack
  - use ⟨class⟩   use object on top of stack

Object creation

- Example:

  Point p = new Point(3) — Java source

  1: new Point

  2: dup

  3: iconst 3

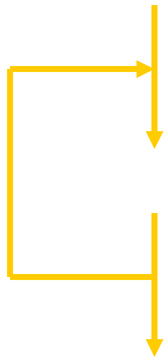
  4: init Point

  (lines 1–4: bytecode)
- No easy pattern to match
- Multiple refs to same uninitialized object
  - Need some form of alias analysis

Alias Analysis

- Other situations:

or



- Equivalence classes based on line where object was created.

Tracking initialize-before-use

- Alias analysis uses line numbers
  - Two pointers to “unitialized object created at line 47” are assumed to point to same object
  - All accessible objects must be initialized before jump backwards (possible loop)
- Oversight in treatment of local subroutines
  - Used in implementation of `try-finally`
  - Object created in `finally` not necessarily initialized
- No clear security consequence
  - Bug fixed

Have proved correctness of modified verifier for init

Bug in Sun's JDK 1.1.4

- Example:

1: jsr 10
2: store 1
3: jsr 10
4: store 2
5: load 2
6: init P
7: load 1
8: use P
9: halt

10: store 0
11: new P
12: ret 0

variables 1 and 2 contain references to two different objects which are both "uninitialized object created on line 11"

Implementing Generics

- Two possible implementations
  - Heterogeneous: instantiate generics
  - Homogeneous: translate generic class to standard class

Idea: replace class parameter <A> by Object, insert casts

Example generic construct: Lists

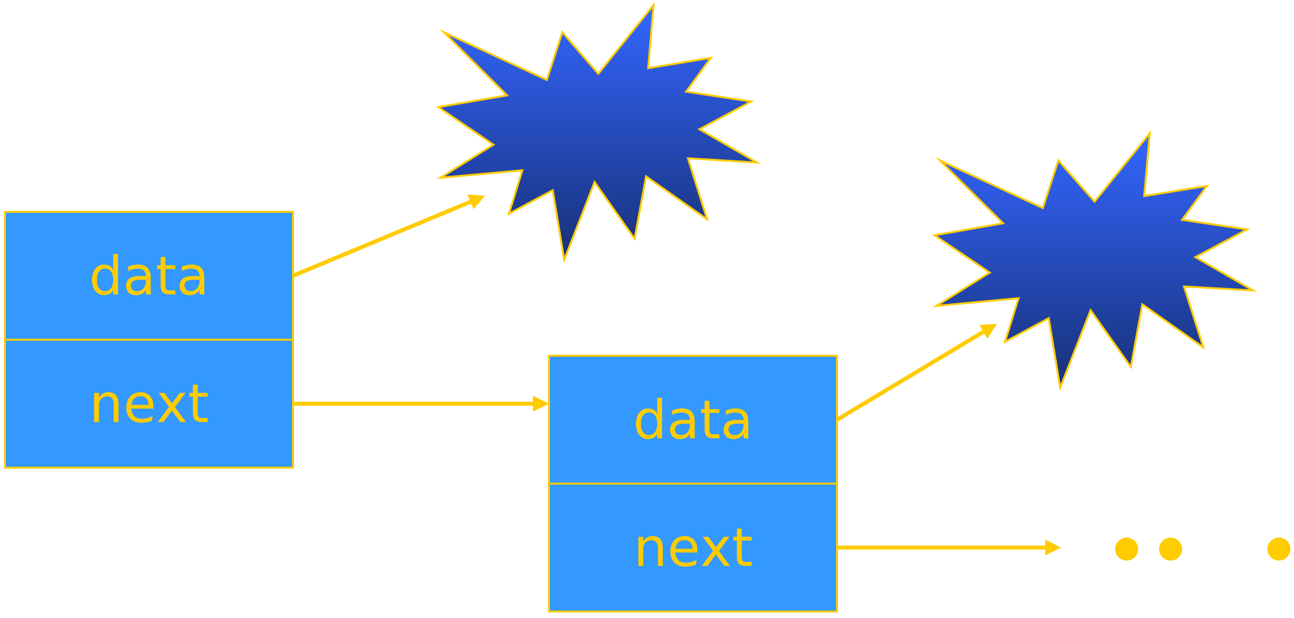
- Lists possible for any type of object
  - For any type t, can have type list_of_t
  - Operations cons, head, tail work for any type
- Define generic list class

```
template <type t> class List {
        private: t* data; List<t> * next;
        public: void      Cons (t* x) { … }
                          t*        Head (     ) { … }
                 List<t>  Tail   (     ) { … }
};
```

Implementation Issues

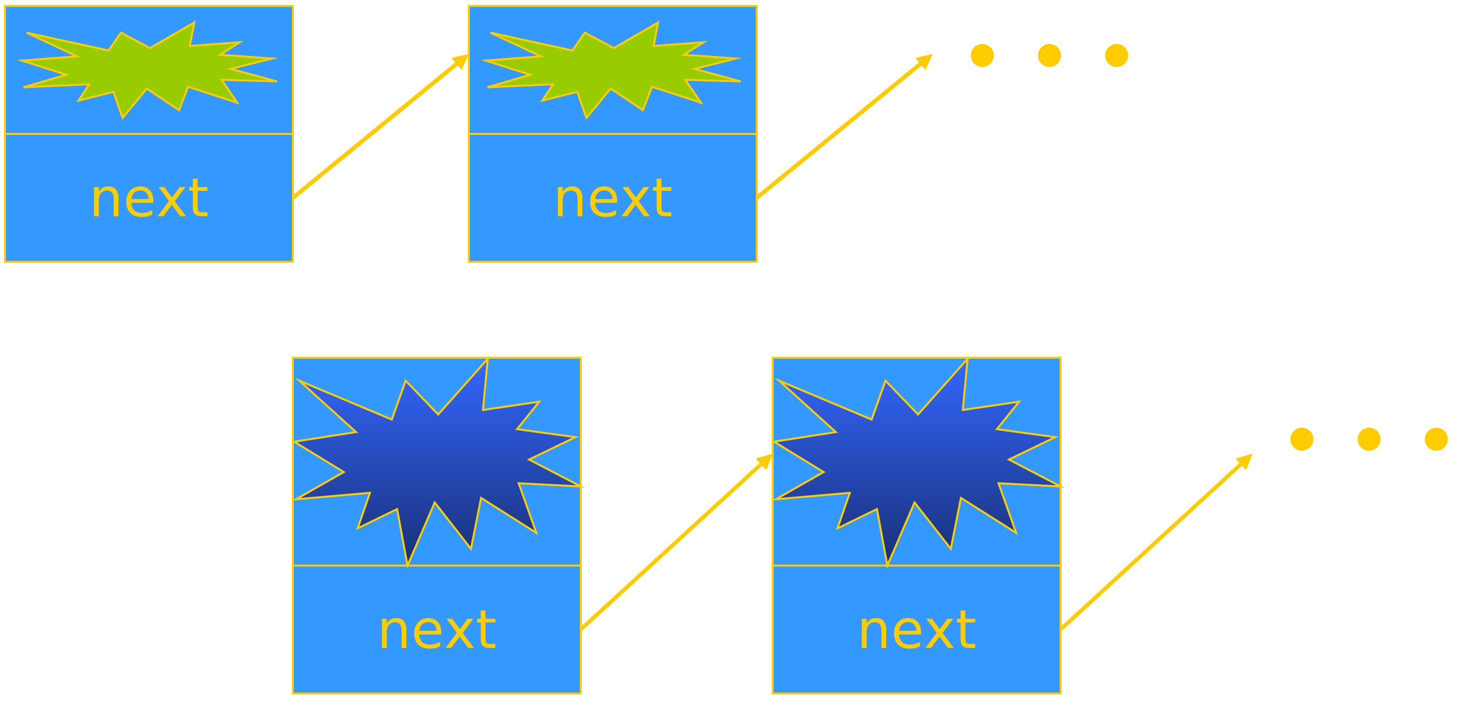
- Data on heap, manipulated by pointer
  - Every list cell has two pointers, data and next
  - All pointers are same size
  - Can use same representation, code for all types
- Data stored in local variables
  - List cell must have space for data
  - Different representation for different types
  - Different code if offset built into code

“Homogeneous Implementation”



Same representation and code for all types of data

“Heterogeneous Implementation”



Specialize representation, code according to type

Example: Hash Table

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

class HashTable < Key implements Hashable, Value> {
    void    Insert (Key k, Value v) {
                int bucket = k.HashCode();
                InsertAt (bucket, k, v);
    }
    ...
};
```

Heterogeneous Implementation

- Compile generic class C<param>
  - Check use of parameter type according to constraints
  - Produce extended form of bytecode class file
    - Store constraints, type parameter names in bytecode file
- Expand when class C<actual> is loaded
  - Replace parameter type by actual class
  - Result is ordinary class file
  - This is a preprocessor to the class loader:
    - No change to the virtual machine
    - No need for additional bytecodes

Generic bytecode with placeholders

```
void Insert (Key k, Value v) {
    int bucket = k.HashCode();
    InsertAt (bucket, k, v);
}
Method void Insert($1, $2)
    aload_1
    invokevirtual #6 <Method $1.HashCode()I>
    istore_3    aload_0    iload_3    aload_1   aload_2
    invokevirtual #7 <Method HashTable<$1,$2>.
                            InsertAt(IL$1;L$2;)V>
    return
```

Instantiation of generic bytecode

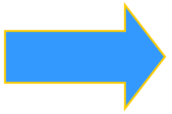
```
void Insert (Key k, Value v) {
    int bucket = k.HashCode();
    InsertAt (bucket, k, v);
}
Method void Insert(Name, Integer)
    aload_1
    invokevirtual #6 <Method Name.HashCode()I>
    istore_3     aload_0    iload_3    aload_1   aload_2
    invokevirtual #7 <Method HashTable<Name,Integer>
                         InsertAt(ILName;LInteger;)V>
    return
```

Load parameterized class file

- Use of invokes loader
- Several preprocess steps
  - Locate bytecode for parameterized class, actual types
  - Check the parameter constraints against actual class
  - Substitute actual type name for parameter type
  - Proceed with verifier, linker as usual.
- Can be implemented with ~500 lines Java code
  - Portable, efficient, no need to change virtual machine

Java 1.5 Solution

- Homogeneous implementation



- Algorithm
  - replace class parameter <A> by Object, insert casts
  - if <A extends B>, replace A by B
- Why choose this implementation?
  - Backward compatibility of distributed bytecode
  - Surprise: faster because class loading is slow

Some details that matter

- Allocation of static variables
  - Heterogeneous: separate copy for each instance
  - Homogenous: one copy shared by all instances
- Constructor of actual class parameter
  - Heterogeneous: class G<T> … T x = new T;
  - Homogenous: new T may just be Object !
- Resolve overloading
  - Heterogeneous: could try to resolve at instantiation time (C++)
  - Homogenous: no information about type parameter
- When is template instantiated?
  - Compile- or link-time (C++)
  - Java alternative: class load time

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
  - Bytecode verifier (example: initialize before use)
  - Implementation of generics
- Security issues

Java Security

- Security
  - Prevent unauthorized use of computational resources
- Java security
  - Java code can read input from careless user or malicious attacker
  - Java code can be transmitted over network –

    code may be *written* by careless friend or malicious attacker

Java is designed to reduce many security risks

Java Security Mechanisms

- Sandboxing
  - Run program in restricted environment
    - Analogy: child's sandbox with only safe toys
  - This term refers to
    - Features of loader, verifier, interpreter that restrict program
    - Java Security Manager, a special object that acts as access control "gatekeeper"
- Code signing
  - Use cryptography to establish origin of class file
    - This info can be used by security manager

Buffer Overflow Attack

- Most prevalent security problem today
  - Approximately 80% of CERT advisories are related to buffer overflow vulnerabilities in OS, other code
- General network-based attack
  - Attacker sends carefully designed network msgs
  - Input causes privileged program (e.g., Sendmail) to do something it was not designed to do
- Does not work in Java
  - Illustrates what Java was designed to prevent

Sample C code to illustrate attack

- Function
  -
  -
- Calling program
  -
  -
  -
- Variations
  -
  -

See: *Smashing the stack for fun and profit*

Java Sandbox

- Four complementary mechanisms
  - Class loader
    - Separate namespaces for separate class loaders
    - Associates *protection domain* with each class
  - Verifier and JVM run-time tests
    - NO unchecked casts or other type errors, NO array overflow
    - Preserves private, protected visibility levels
  - Security Manager
    - Called by library functions to decide if request is allowed
    - Uses protection domain associated with code, user policy
    - Recall: stack inspection problem on midterm

Why is typing a security feature?

- Sandbox mechanisms all rely on type safety
- Example
  - Unchecked C cast lets code make any system call

```
int (*fp)()     /* variable "fp" is a function pointer
   */
...
fp = addr;      /* assign address stored in an integer var
    */
(*fp)(n);       /* call the function at this address
   */
```

Other examples involving type confusion in book

Security Manager

- Java library functions call security manager
- Security manager object answers at run time
  - Decide if calling code is allowed to do operation
  - Examine protection domain of calling class
    - Signer: organization that signed code before loading
    - Location: URL where the Java classes came from
  - Uses the system policy to decide access permission

Sample SecurityManager methods

Stack Inspection

- Permission depends on
  - Permission of calling method
  - Permission of all methods above it on stack
    - Up to method that is trusted and asserts this trust

Many details omitted here

Stories: Netscape font / passwd bug; Shockwave plug-in

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
  - Bytcode 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

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;}

}
```