Programación orientada a objetos: Java – implementación

Paradigmas de la Programación
FaMAF-UNC 2015
capítulo 13
basado en filminas de John Mitchell

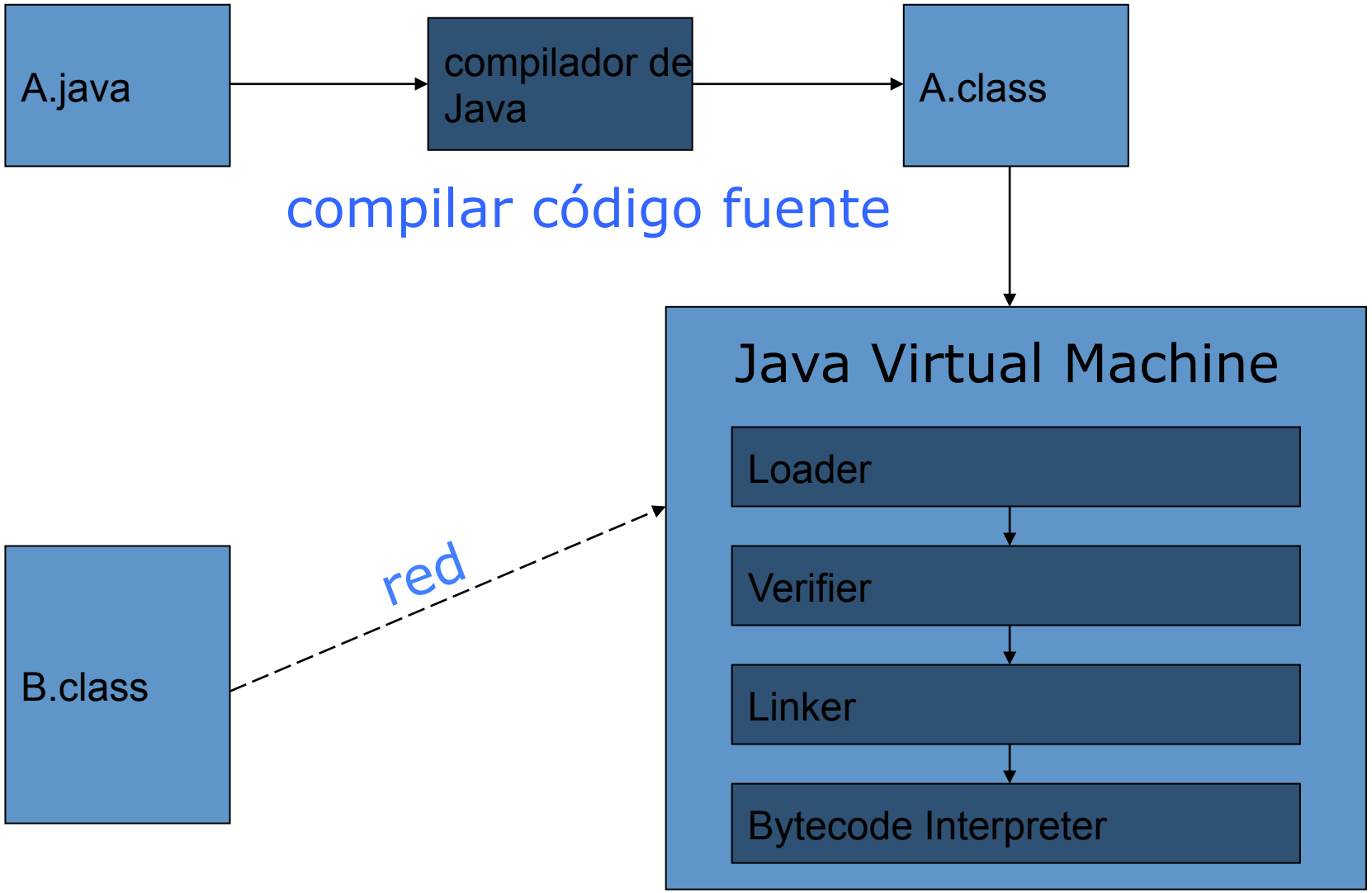
contenidos

- objetos en Java
  - Clases, encapsulación, herencia
- sistema de tipos
  - tipos primitivos, interfaces, arreglos, excepciones
- genéricos (añadidos en Java 1.5)
  - básicos, wildcards, …
- máquina virtual
  - Loader, verifier, linker, interpreter
  - Bytecodes para lookup de métodos
- temas de seguridad

implementación

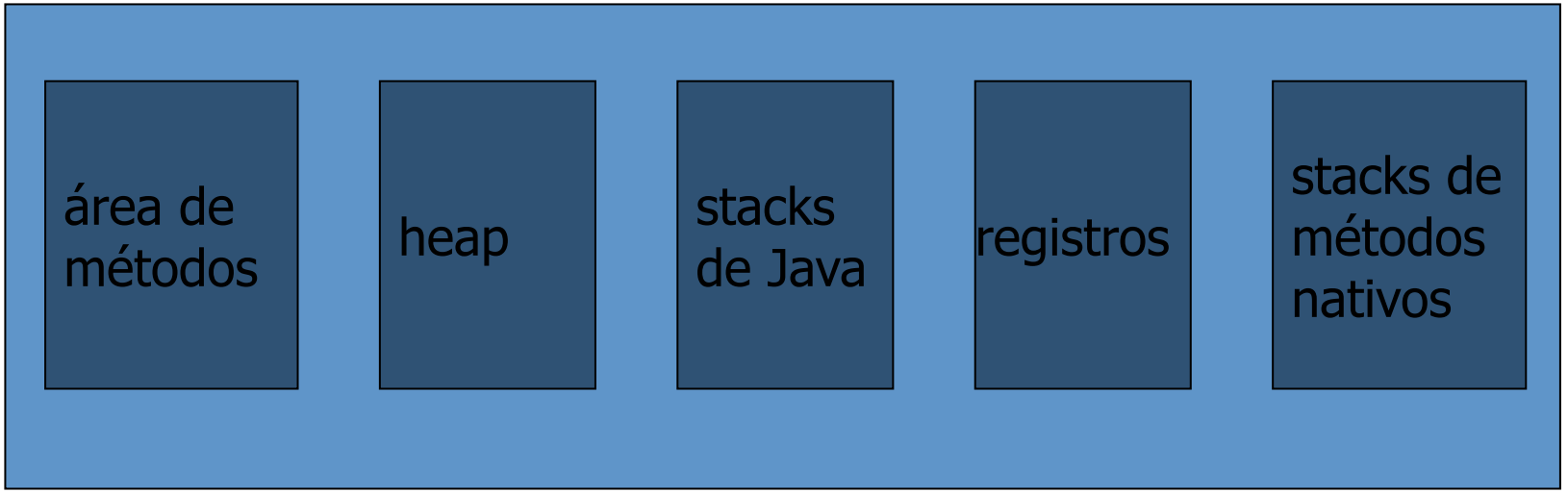
- compilador y máquina virtual
  - el compilador produce bytecode
  - la máquina virtual carga clases a demanda, verifica propiedades del bytecode e interpreta el bytecode
- por qué este diseño?
  - ya se habían usado intérpretes / compiladores de bytecode antes: Pascal, Smalltalk
  - minimizan la parte de la implementación dependiente de máquina
    - la optimización se hace en el bytecode
    - se mantiene muy simple el intérprete de bytecode
  - para Java, también aporta portabilidad
    - se puede transmitir el bytecode por la red

Arquitectura de la JVM



áreas de memoria de la JVM

- el programa en Java program tiene uno o más threads
- cada thread tiene su propio stack
- todos los threads comparten el heap



carga de clases

- el sistema de ejecución carga las clases a medida que se necesitan
  - cuando se referencia una clase, el sistema de carga busca el archivo de instrucciones de bytecode compiladas
- el mecanismo de carga por defecto se puede sustituir definiendo otro objeto ClassLoader
  - se extiende la clase ClassLoader
  - ClassLoader no implementa el método abstracto loadClass, sino que tiene métodos que pueden usarse para implementar loadClass
  - se pueden obtener bytecodes de otra fuente
    - la VM restringe la comunicación entre applets al sitio que provee el 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

linker y verificador de la JVM

- Linker
  - añade la clase o interfaz compiladas al sistema de ejecución
  - crea los campos estáticos y los inicializa
  - resuelve nombres, reemplazándolos con referencias directas
- Verificador
  - comprueba el bytecode de una clase o interfaz antes de que se cargue
  - lanza la excepción VerifyError

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

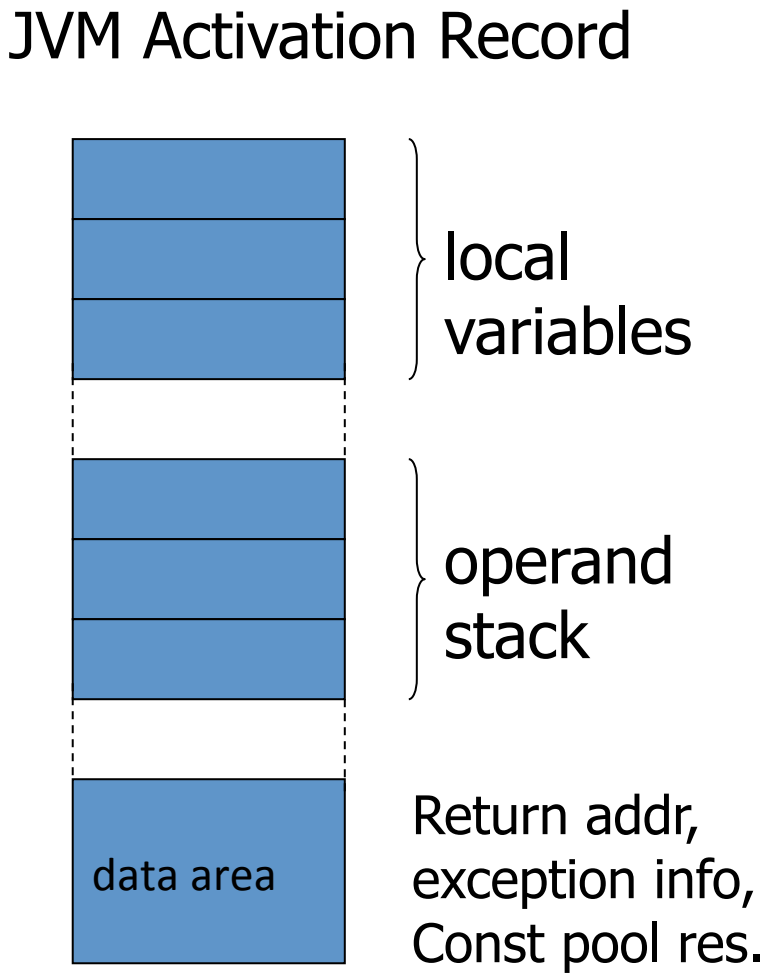
- Java

```
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



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?

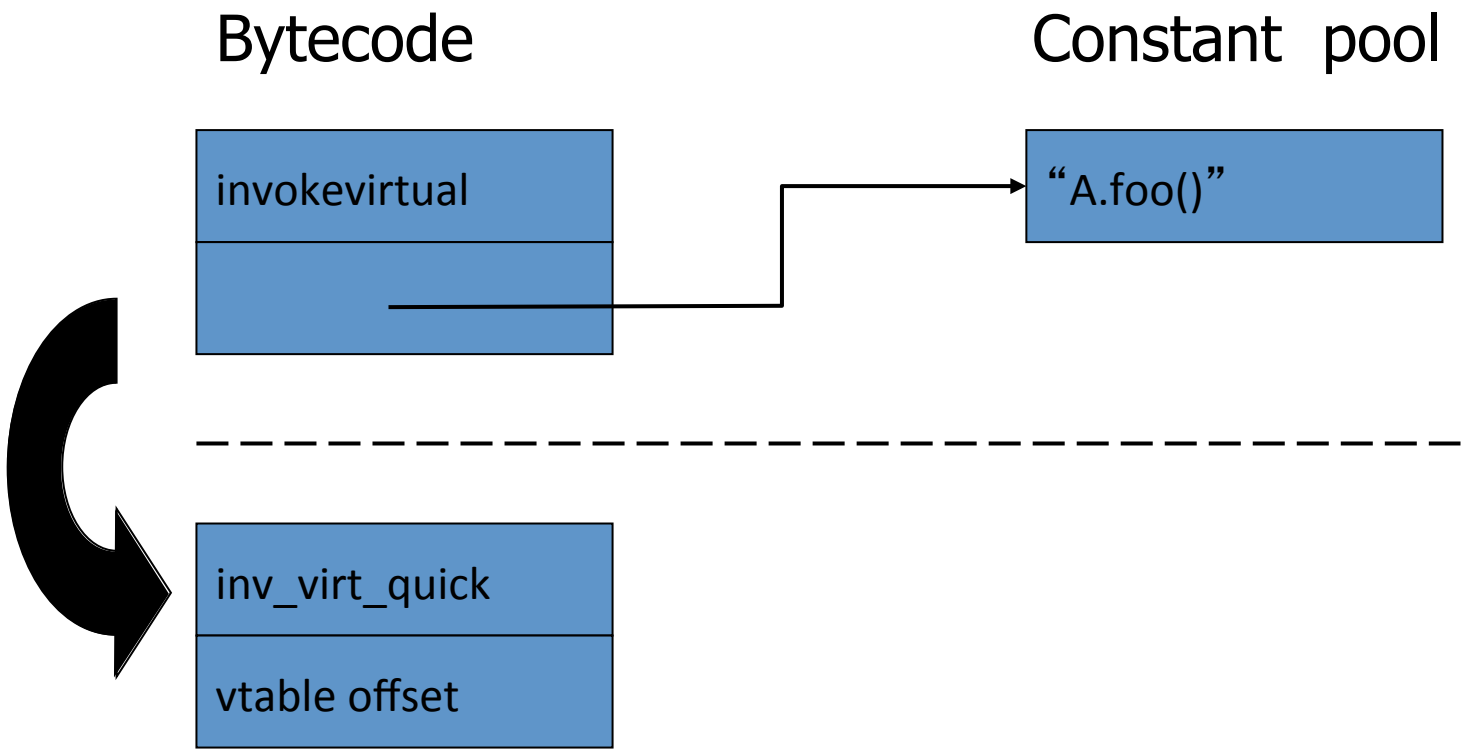
```
interface A {
   public void f();
}
interface B {
   public void g();
}
class C implements A, B {
   …;
}
```

Class C cannot have method f first *and* method g first

invokevirtual <method-spec>

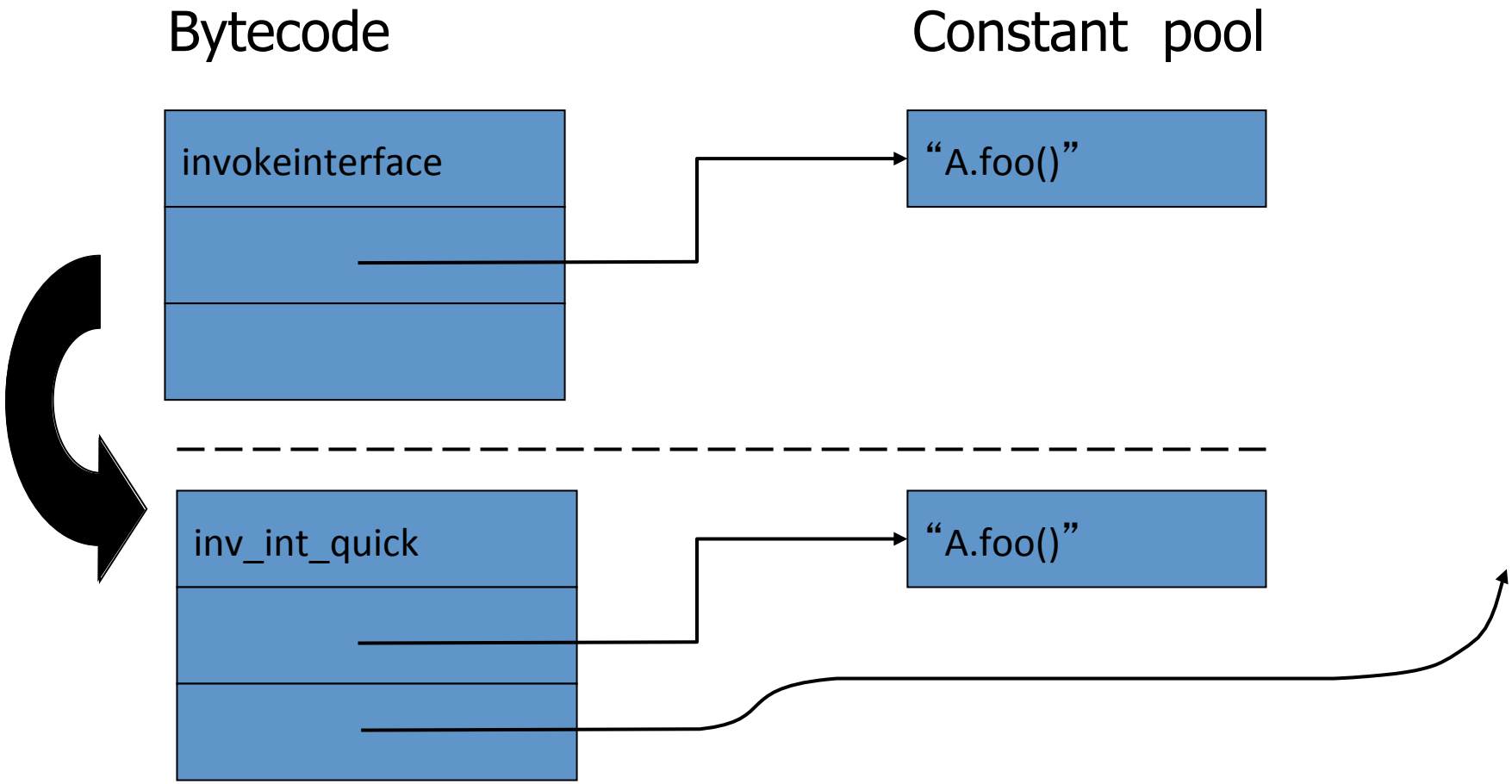
- Similar to invokeinterface, but class is known
- Search for method
  - search the method table of this class
  - find method with the given name and signature
- Can we use static type for efficiency?
  - Each execution of an instruction will be to object from subclass of statically-known class
  - 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

    (idealization for purpose of presentation)

Object creation

- Example:

  Point p = new Point(3) Java source

  1: new Point
  2: dup
  3: iconst 3
  4: init Point

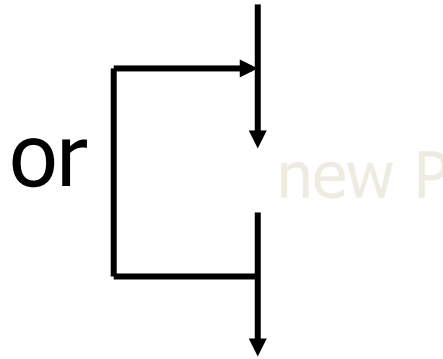
  bytecode

- No easy pattern to match
- Multiple refs to same uninitialized object
  - Need some form of alias analysis

Alias Analysis

- Other situations:

1: new P
2: new P
3: init P



- 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 early 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

Aside: bytecodes for try-finally

- Idea
  - Finally clause implemented as lightweight subroutine
- Example code

```
static int f(boolean bVal) {
        try {
            if (bVal) { return 1; }
            return 0;
         }
         finally {
              System.out.println(“About to return");
 } }
```

- Bytecode on next slide
  - Print before returning, regardless of which return is executed

Bytecode

(from http://www.javaworld.com/javaworld/jw-01-1997/jw-02-hood.html?page=2)

```
0 iload_0       // Push local variable 0 (arg passed as divisor)
1 ifeq 11       // Push local variable 1 (arg passed as dividend)
4 iconst_1      // Push int 1
5 istore_3      // Pop an int  (the 1), store into local variable 3
6 jsr 24        // Jump to the mini-subroutine for the finally clause
9 iload_3       // Push local variable 3 (the 1)
10 ireturn      // Return int on top of the stack (the 1)
.
.
.
24 astore_2     // Pop the return address, store it in local variable 2
25 getstatic #8 // Get a reference to java.lang.System.out
28 ldc #1       // Push <String "About to return."> from the constant pool
30 invokevirtual #7 // Invoke System.out.println()
33 ret 2        // Return to return address stored in local variable 2
```

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"

Bytecode verifier not designed for code that creates uninitialized object in jsr subroutine

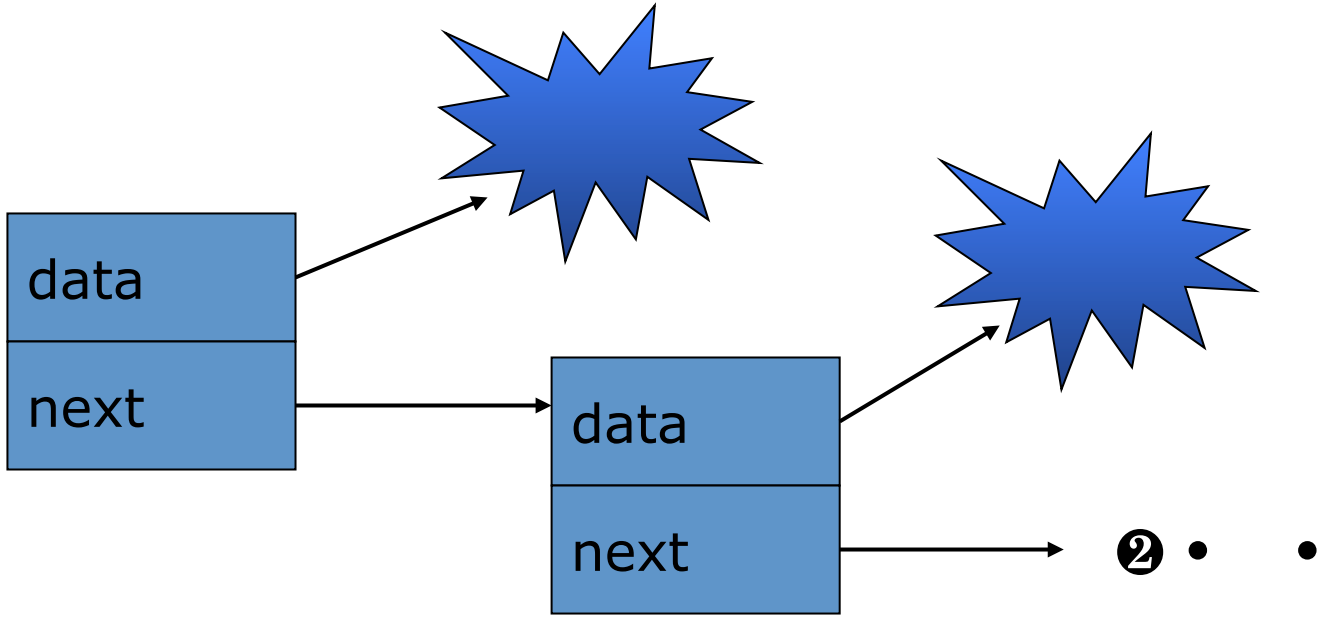
Implementing Generics

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

- Example for next few slides: generic list class

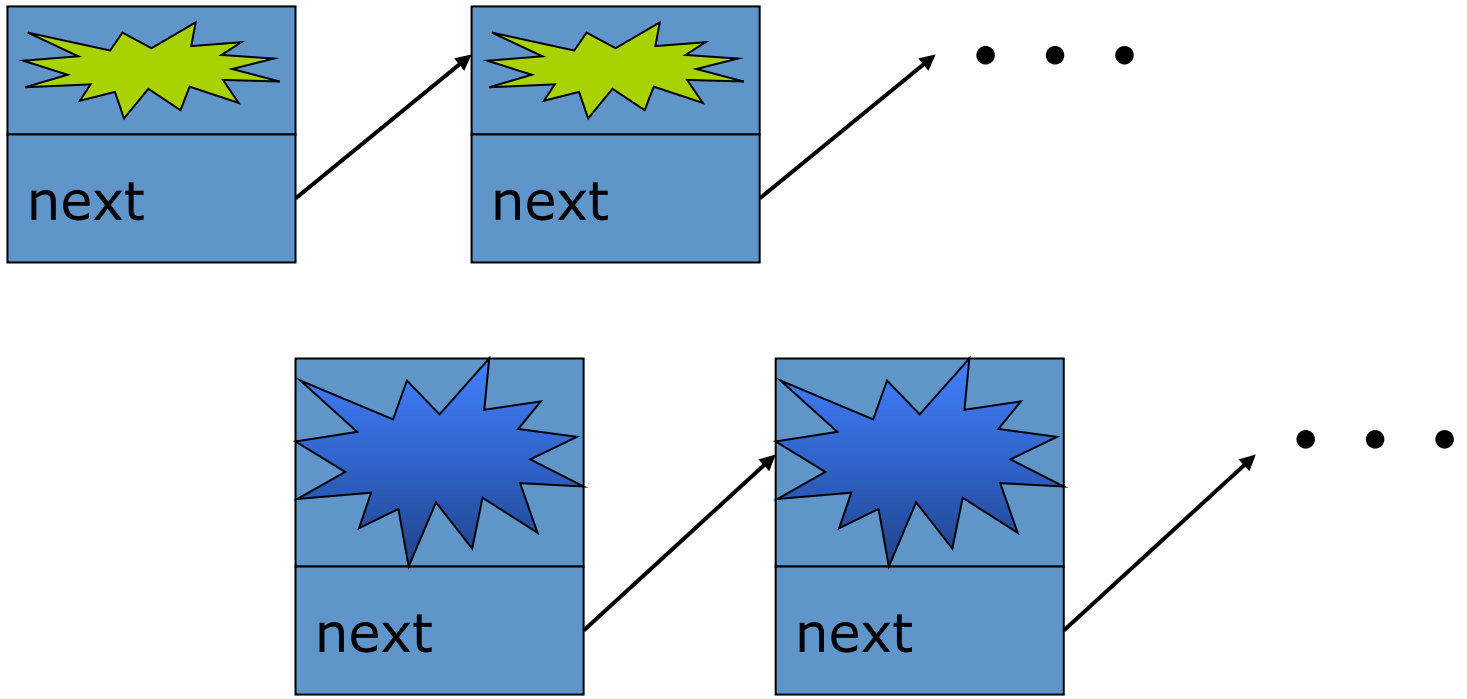
```
template <type t> class List {
    private: t* data; List<t> * next;
    public: void      Cons (t* x) { … }
            t*        Head (    ) { … }
                List<t>  Tail   (    ) { … }
};
```

“Homogeneous Implementation”



Same representation and code for all types of data

“Heterogeneous Implementation”



Specialize representation, code according to type

Issues

- Data on heap, manipulated by pointer (Java)
  - 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 (C++)
  - List cell must have space for data
  - Different representation for different types
  - Different code if offset of fields built into code
- When is template instantiated?
  - Compile- or link-time (C++)
  - Java alternative: class load time – next few slides
  - Java Generics: no “instantiation”, but erasure at compile time
  - C# : just-in-time instantiation, with some code-sharing tricks …

Heterogeneous Implementation for Java

- 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

A heterogeneous implementation is possible, but was not adopted for standard

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);
    }
    …
};
```

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
```

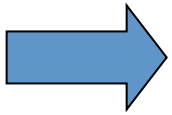
Loading parameterized class file

- Use of HashTable <Name, Integer> 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 Implementation

- Homogeneous implementation

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



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

- 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: sometimes faster because class loading 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 !
    - Creation of new object is not allowed in Java
- Resolve overloading
  - Heterogeneous: resolve at instantiation time (C++)
  - Homogenous: no information about type parameter

Example

- This Code is not legal java
  - class C<A> { A id (A x) {...} }
  - class D extends C<String> {
    Object id(Object x) {...}
    }
- Why?
  - Subclass method looks like a different method, but after erasure the signatures are the same

contenidos

- objetos en Java
  - Clases, encapsulación, herencia
- sistema de tipos
  - tipos primitivos, interfaces, arreglos, excepciones
- genéricos (añadidos en Java 1.5)
  - básicos, wildcards, …
- máquina virtual
  - Loader, verifier, linker, interpreter
  - Bytecodes para lookup de métodos
- temas de seguridad

seguridad en Java

- seguridad
  - evitar uso no autorizado de recursos computacionales
- seguridad en Java
  - el código Java puede leer input de usuarios despistados o atacantes maliciosos
  - el código Java se puede transmitir por la red

Java está diseñado para reducir riesgos de seguridad

mecanismos de seguridad

- Sandboxing (jugar en el arenero)
  - el programa se ejecuta en un entorno restringido
  - se aplica a:
    - características del loader, verificador, e intérprete que restringen al programa
    - Java Security Manager, un objeto especial que ejerce control de acceso
- firma de código
  - se usan principios criptográficos para establecer el origen de un archivo de clase
  - la usa el security manager

ataque de Buffer Overflow

- es el problema de seguridad más frecuente
- en general, basado en red:
  - el atacante envía mensajes de red diseñados especialmente
  - el input hace que un programa con privilegios (por ej., Sendmail) haga algo que no tenía que hacer
- no funciona en Java!

ejemplo de código en C para ataque de buffer overflow

```
void f (char *str) {
    char buffer[16];
    …
    strcpy(buffer,str);
}
void main() {
    char large_string[256];
    int i;
    for( i = 0; i < 255; i+
  +)
   large_string[i] = 'A';
    f(large_string);
}
```

- la función
  - copia str a un buffer hasta que se encuentra el caracter nulo
  - podría escribir hasta pasado el final del buffer, por encima de la dirección de retorno de la función!!
- la llamada
  - escribe 'A' sobre el activation record de f
  - la función “retorna” a la ubicación 0x4141414141
  - esto causa un segmentation fault
- variaciones
  - poner una dirección con significado en el string
  - poner código en el string y saltar ahí!

para saber más: *Smashing the stack for fun and profit*

Java Sandbox

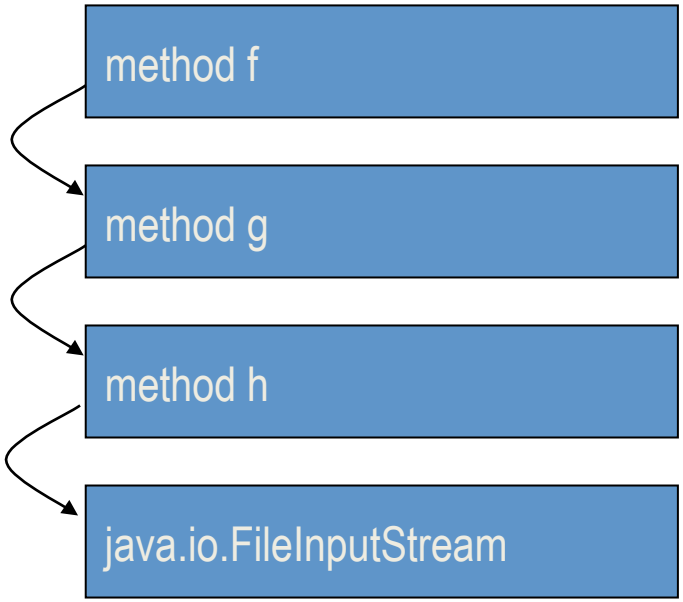
- cuatro mecanismos complementarios
  - Class loader
    - namespaces distintos para distintos class loaders
    - asocia un *protection domain* con cada clase
  - tests en tiempo de ejecución del Verifier y JVM
    - no se permiten casteos sin comprobación de tipos ni otros errores de tipo, no se permite array overflow
    - preserva los niveles de visibilidad private y protected
  - Security Manager
    - lo llaman las funciones para decidir si deben hacer lugar a un pedido
    - usa el *protection domain* asociado al código y política de usuario
    - inspección del stack

Security Manager

- las funciones de la biblioteca de Java llaman al security manager
- respuesta en tiempo de ejecución
  - decide si el código que llama tiene permiso para hacer la operación
  - examinar el dominio de protección de la clase que llama
    - Signer: organización que firmó el código antes de cargarlo
    - Ubicación: URL de donde vienen las clases
  - da permiso de acceso según la política del sistema

inspección del stack

- el permiso depende de:
  - permiso del método que llama
  - permiso de todos los métodos por encima de él en el stack, hasta llegar a un método confiable



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

ejemplos de métodos del Security Manager

checkExec	comprueba si los comandos de sistema se pueden ejecutar.
checkRead	comprueba si un archivo se puede leer.
checkWrite	comprueba si un archivo se puede escribir.
checkListen	comprueba si un puerto determinado se puede escuchar.
checkConnect	comprueba si se puede crear una conexión de red.
checkCreate ClassLoader	comprueba para evitar que se instalen más ClassLoaders.

resumen

- objetos
  - tienen campos y métodos
  - alojados en el heap, se acceden con punteros, con recolección de basura
- clases
  - Public, Private, Protected, Package (no exactamente como en C++)
  - pueden tener miembros estáticos (propios de la clase)
  - Constructores y métodos finalize
- herencia
  - herencia simple
  - métodos y clases finales (no pueden tener hijas)

resumen

- subtipado
  - determinado por la jerarquía de herencia
  - una clase puede implementar muchas interfaces
- Virtual machine
  - carga bytecode para clases en tiempo de ejecución
  - el verificador comprueba el bytecode
  - el intérprete también hace comprobaciones en tiempo de ejecución
    - casteos
    - límites de arreglos
- portabilidad y seguridad

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