Laboratorio di Programmazione 3

Lezioni 18, 19 e 20: Portabilità e Sicurezza, Java (cap. 13)

Massimo Tivoli

Versione modificata delle slides di J. C. Mitchell fornite con il libro "Concepts in Programming Languages" come materiale di supporto alla didattica

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 accessed through pointers (they are all on the heap) and pointer assignment is the only form of object assignment
  - No functions, no multiple inheritance, no *operator overloading*, few automatic coercions (*much simpler than C++*)
  - Always pass parameters to Java methods by value (*pass-by-value*)
- Portability and network transfer
  - Bytecode interpreter on many platforms (*the JVM*)
  - Dynamic linking
- Reliability and Safety
  - Typed source and typed bytecode language (compile-time type checks by the compiler)
  - Run-time type and bounds checks (by the JVM)
  - Garbage collection

You can see that many of the above decisions decrease efficiency.

As an excerpt from "Concepts in Programming Languages" – Chapter 13

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
  - Language interoperability
    - Provision for “native” methods (efficiency and access to legacy utilities)

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 - as in other languages
- Field – data member
- Method - member function
- Static members - class fields and methods
- this - self
- Package - set of classes in shared namespace
- Native method - method written in another language, often C

As an excerpt from "Concepts in Programming Languages" – Chapter 13

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

Point Class

```
class Point {
    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

Static fields and methods

```
class Pippo {
  static int x;
  static { /* code to be executed once, when class is loaded */}
}
```

- They can access only static fields and other static methods.
- Some other restrictions, e.g., a static block cannot raise an exception

An example:

```
public class MyProgramMainClass {
  …
  public static void main(String[] args) {…} …}
```

As an excerpt from “Concepts in Programming Languages” – Chapter 13

Garbage Collection and Finalize

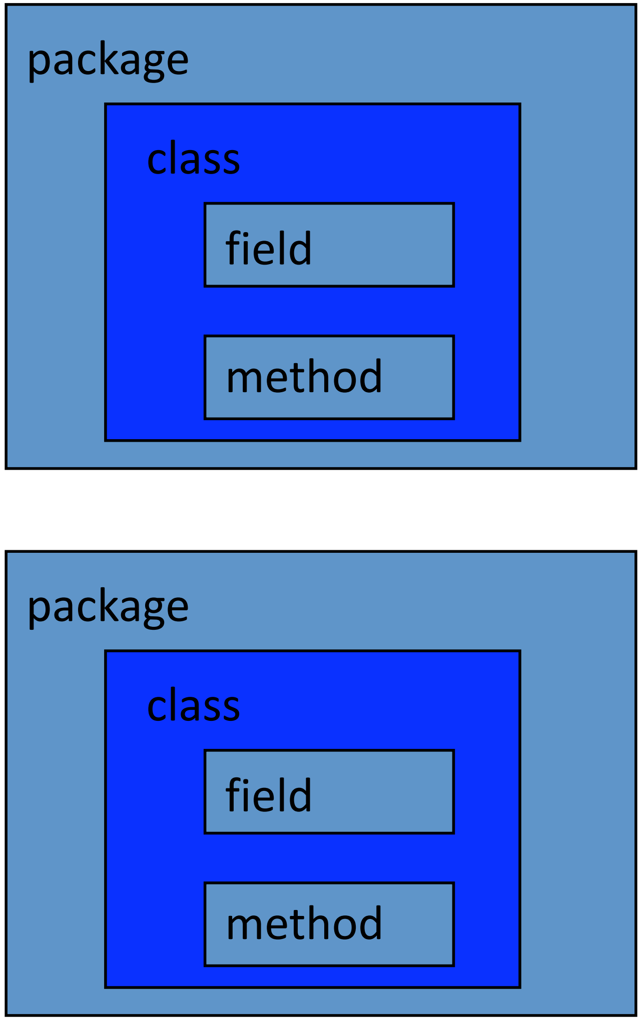
- Objects are garbage collected
  - No explicit *free*
  - Avoids dangling pointers and resulting type errors
- Problem
  - What if object has opened file or holds lock?
- Solution
  - *finalize* method, called by the garbage collector
    - Before space is reclaimed, or when virtual machine exits
    - portant convention: call super.finalize

```
class … {…
  protected void finalize () { super.finalize(); close(file);}};
```

As an excerpt from "Concepts in Programming Languages" – Chapter 13

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



As an excerpt from "Concepts in Programming Languages" – Chapter 13

Visibility and access

- Four visibility distinctions
  - public: accessible anywhere the class is visible
  - protected: accessible to methods of the class and any subclasses, as well as to other classes in the same package,
  - private: accessible only in the class itself,
  - package (default visibility): accessible only to code in the same package (not visible to subclasses in other packages)
- In other words, 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)
  - $\underbrace{\text{java.lang}}_{\text{package}}.\underbrace{\text{String}}_{\text{class}}.\underbrace{\text{substring()}}_{\text{method}}$

As an excerpt from "Concepts in Programming Languages" – Chapter 13

Inheritance

- Similar to Smalltalk, C++
  - method overriding (keyword *super*) and field hiding
  - constructors (keyword *super,* implicitly by the programmer or explicitly by the compiler… different from the *finalize* case)
- *Subclass* inherits from *superclass*
  - Single inheritance only (but Java has interfaces… see later)
  - *Interface* different from *abstract class*, no names clash… a more clean feature
- Some additional features
  - Conventions regarding *super* in constructor and *finalize* methods
  - Final classes and methods
    - Singleton pattern, java.lang.System class, …

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
 };
```

As an excerpt from “Concepts in Programming Languages” – Chapter 13

Class *Object*

- Every class extends another class
  - Superclass is *Object* if no other class named
- Methods of class *Object* (is the one class that has no superclasses)
  - GetClass – return the Class object representing class of the object, used for reflection purposes
  - ToString – returns string representation of object
  - equals – default object equality (not ptr equality)
  - hashCode – returns an integer that can be used to store the object in a has table
  - clone – makes a duplicate of an object
  - wait, notify, notifyAll – used with concurrency
  - finalize – which is run just before an object is destroyed
- Every object that we create has at least the above methods

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

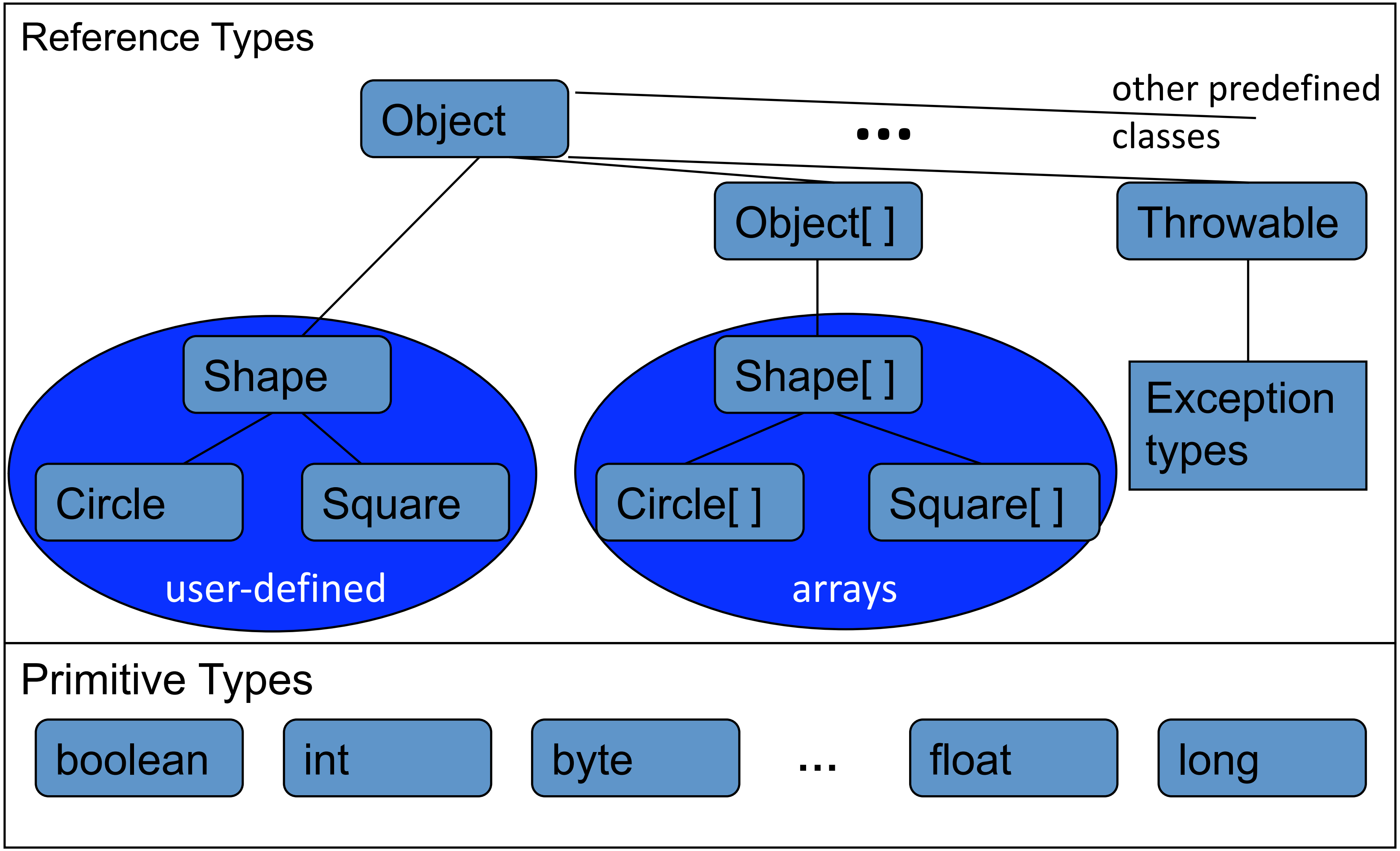
- Objects in Java
  - Classes, encapsulation, inheritance
- Type system
  - Primitive types, interfaces, arrays, exceptions
- Generics (added in Java 1.5)
  - Basics, wildcards, …
- Virtual machine
  - Loader, verifier, linker, interpreter
  - Bytecodes for method lookup
- Security issues

Java Types

- Two general kinds of types
  - 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
    - If A x, then can use x as argument to method that requires B
    - If B x, then can try to cast x to A
    - Downcast checked at run-time, may raise exception

As an excerpt from "Concepts in Programming Languages" – Chapter 13

Classification of Java types



As an excerpt from "Concepts in Programming Languages" – Chapter 13

Subtyping

(unique form of sub-classing in Java, no inheritance without subtyping as in C++ through private base classes)

- Primitive types
  - Conversions: int -> long, double -> long, …
  - e.g., *Object* objects in a *LinkedList* – type cast checked at run-time, it raises an exception if the referenced object does not have the designated type
- 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 (implements, extends)
    - It allows objects to support (multiple) common behaviors without sharing any common implementation
- Arrays
  - Covariant subtyping – not consistent with semantic principles… see later

As an excerpt from "Concepts in Programming Languages" – Chapter 13

Java class subtyping

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

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
}
```

As an excerpt from "Concepts in Programming Languages" – Chapter 13

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)

    More about this later …

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 are 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 (e.g., Circle[])

Array subtyping

- Covariance
  - if S <: T then S[ ] <: T[ ] (this is just a design choice)
- Standard type error (*array covariance problem*)

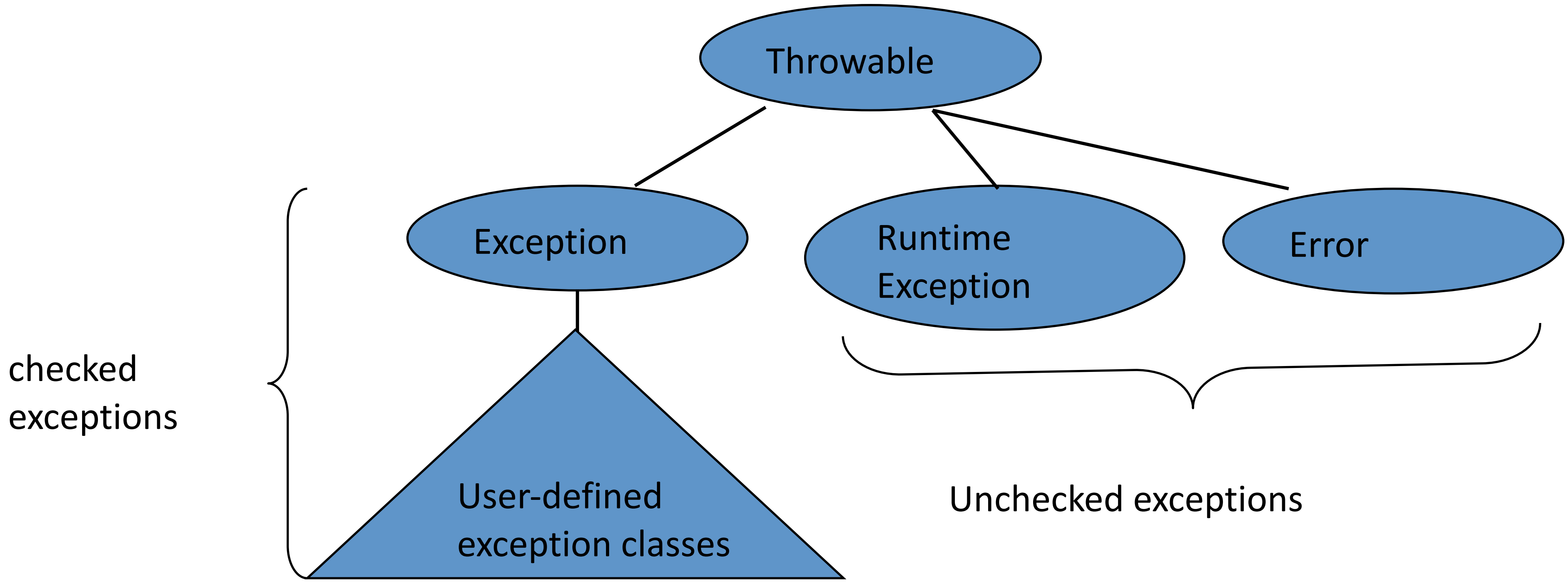
```
class A {…}
class B extends A {…}
B[ ] bArray = new B[10]
A[ ] aArray = bArray      // considered OK since B[] <: A[]
aArray[0] = new A()       // compiles, but run-time error
                          // raises ArrayStoreException
```

As an excerpt from "Concepts in Programming Languages" – Chapter 13

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 (all subtypes of *Throwable*)
    - <u>advantage</u>: useful info is stored in the exception object hence allowing one to carry information from the point in which the exception has been thrown to the handler that catches it
  - Subtyping between exception classes
    - Use subtyping to match type of exception or pass it on …
    - Similar functionality to ML pattern matching in handler
    - <u>A handler matches the class of an exception if it names either the same class or a superclass</u>
  - 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

As an excerpt from "Concepts in Programming Languages" – Chapter 13

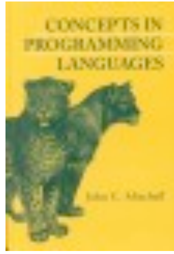
Try/finally blocks

- Exceptions are caught in try blocks

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

Lesson learned ☺ :

read carefully



try through the Java compiler

As an excerpt from "Concepts in Programming Languages" – Chapter 13

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 did not have templates
  - No parametric polymorphism
  - Many considered 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 1.0?

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();
```

a run-time check is needed

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

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

no run-time check

clarity

As an excerpt from "Concepts in Programming Languages" – Chapter 13

Why no generics in early Java ?

- Many proposals
- Basic language goals seem clear
- Details take some effort to work out
  – easy implementation (homogeneous implementation)
    - from Java with templates to Java without templates
    - efficiency decreases, many run-time-checked type conversions
  – alternative implementation
    - compile generic class in a form of class file that has type parameters and load the instantiation of the generic class by instantiating this class file at class-load time
    - more efficient code, but more classes to be loaded (class loading is slow)

Java Community proposal (JSR 14) incorporated into Java 1.5

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

As an excerpt from "Concepts in Programming Languages" – Chapter 13

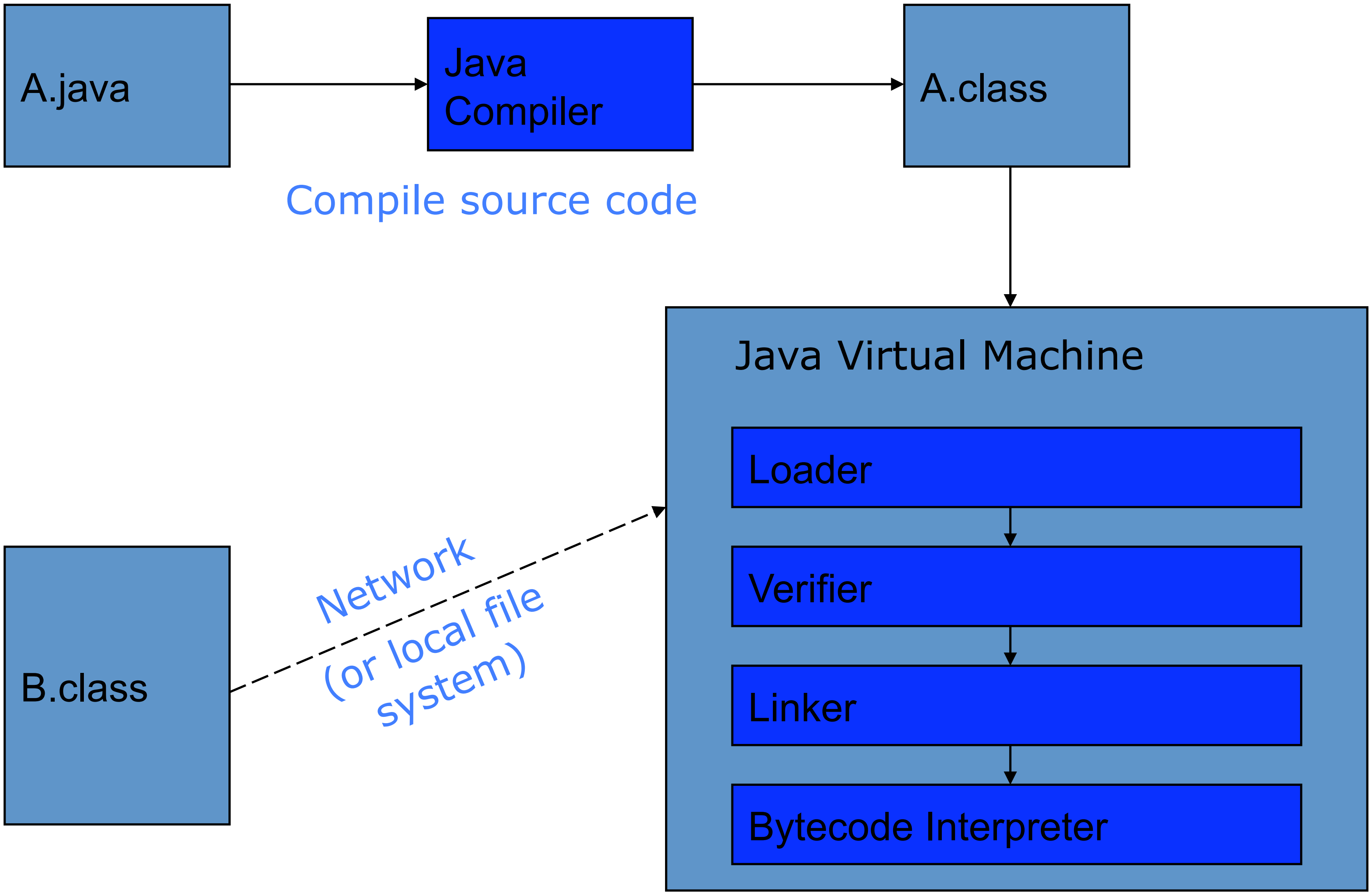
Outline

- 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



As an excerpt from "Concepts in Programming Languages" – Chapter 13

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/remote source

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

- 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 a superclass of the object is known at compile-time
  - invokeinterface - when only an interface of the object is known at compile-time
  - invokestatic - static methods
  - invokespecial - some special case (not discussed)

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

As an excerpt from "Concepts in Programming Languages" – Chapter 13

JVM uses stack machine

- Java

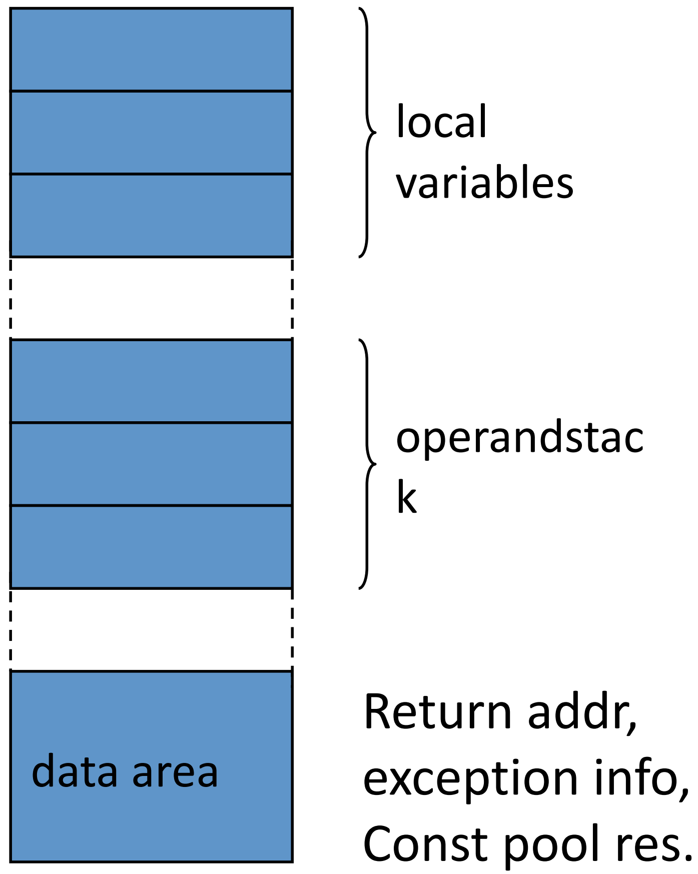
```
Class A extends Object {
    int i
    void f(int val) { i = val + 1;}
}
```

- 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
  - E.g., getfield #18 <Field Obj var>
- Second execution
  - Use modified "quick" instruction to simplify search
  - getfield_quick 6 (*the field has been found and it was located 6 bytes below the first location of the object*)

As an excerpt from "Concepts in Programming Languages" – Chapter 13

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?

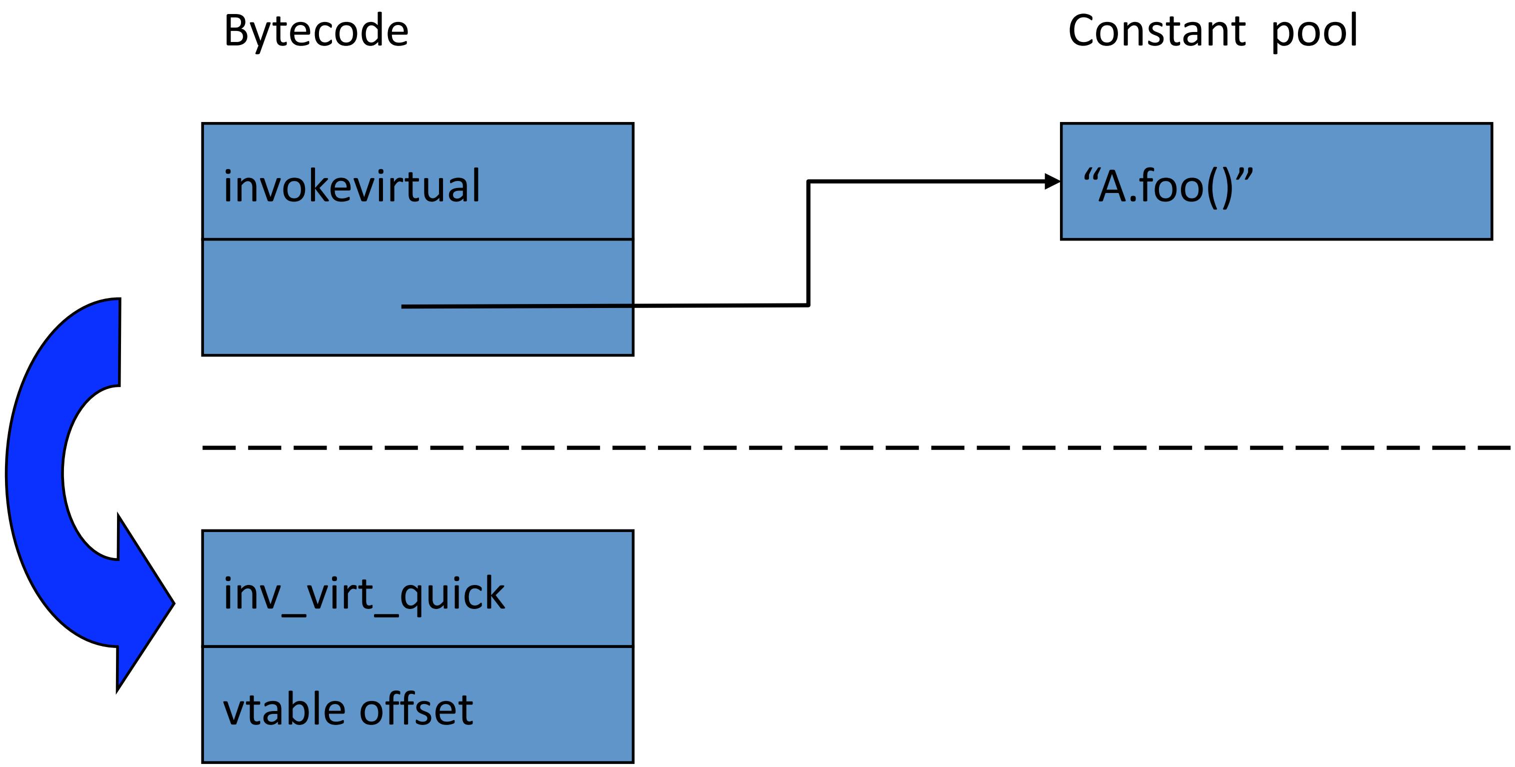
```
interface Incrementable {
    public void inc();
}
class IntCounter implements Incrementable {
    public void add(int);
    public void inc();
   public int value();
}
class FloatCounter implements Incrementable {
    public void inc();
    public void add(float);
    public float value();
}
```

As an excerpt from "Concepts in Programming Languages" – Chapter 13

invokevirtual <method-spec>

- Similar to invokeinterface, but the superclass is known at compile-time
- 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

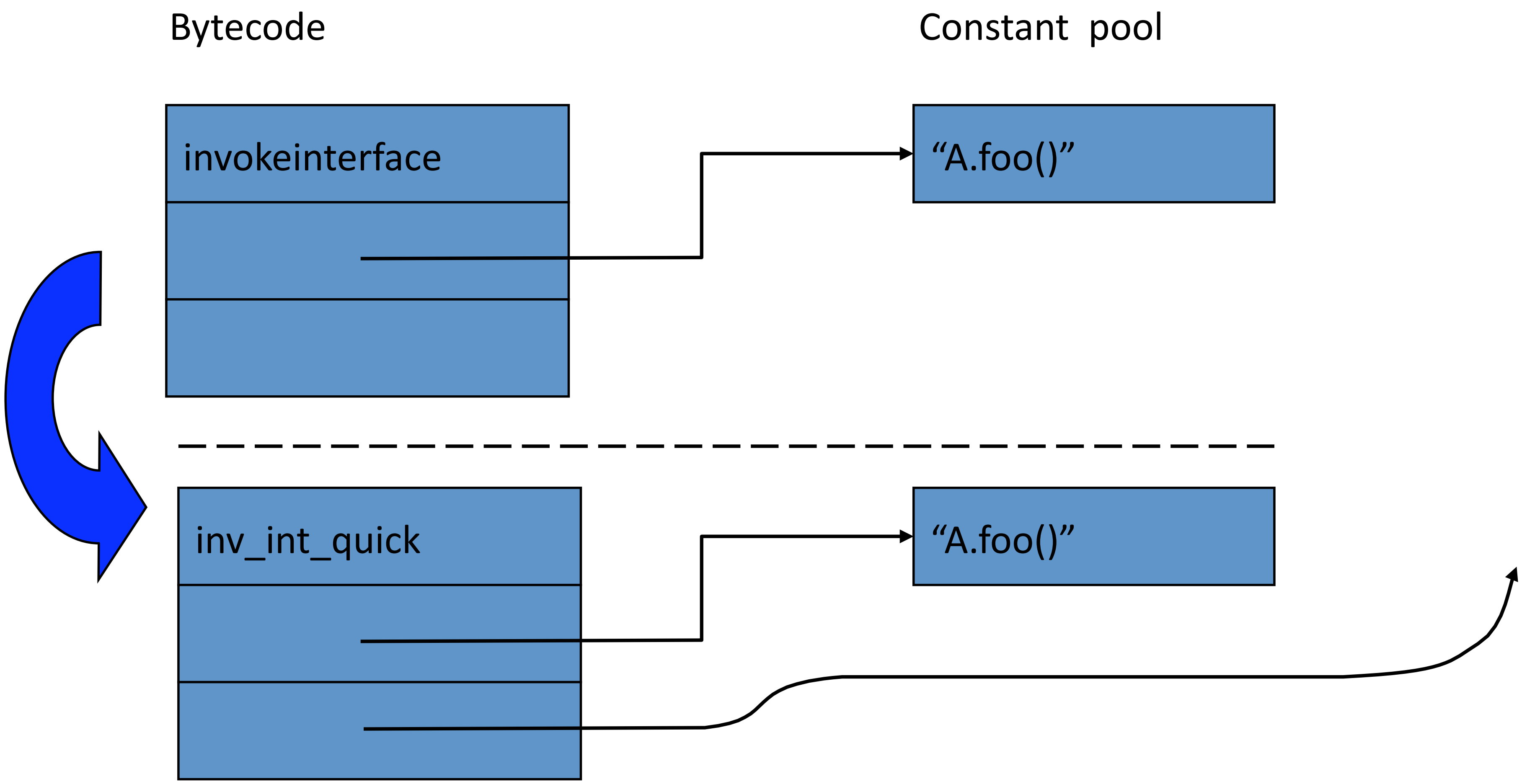
Bytecode rewriting: invokevirtual



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

As an excerpt from "Concepts in Programming Languages" – Chapter 13

Bytecode rewriting: invokeinterface



Cache address of method; check class on second use

As an excerpt from "Concepts in Programming Languages" – Chapter 13

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
  - E.g., Sendmail always runs as super-user process
- 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
    - when bytecode is executed, some operations that can be written in Java language might not be allowed to proceed, just the way that, when a child plays in a sandbox, an adult supervisory may give the child only those toys that the supervisor considers safe
  - 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
    - E.g., Only some code producers are allowed to execute code, different rights for different code producers

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

```
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);
}
```

- Function
  - Copies str into buffer until null character found
  - Could write past end of buffer, *over function retun addr*
- Calling program
  - Writes 'A' over f activation record
  - Function f "returns" to location 0x4141414141
  - This causes segmentation fault
- Variations
  - Put meaningful address in string
  - Put *code* in string and jump to it !!

Not so simple in practice… it requires several attempts by the attacker

As an excerpt from "Concepts in Programming Languages" – Chapter 13

Java Sandbox

- Four complementary mechanisms to restrict the operations of a Java bytecode
  - Class loader
    - Separation between trusted and untrusted class libraries by making it possible to load each with different class loaders
    - Separate namespaces for classes loaded by different class loaders
    - Place code into categories (associates *protection domain* with each class) that let the security manager to restrict the actions that specific code will be allowed to take
  - Verifier and JVM run-time tests
    - NO unchecked casts or other type errors, NO array overflow
      - no method will overflow the operand stack
      - no illegal data conversions (e.g., an integer to a pointer)
      - all method are called with parameter of correct type (type-confusion attacks prevention, discussed later)
    - Preserves private, protected visibility levels
  - Security Manager
    - Uses protection domain associated with code, user policy
      - code signer + class location

As an excerpt from “Concepts in Programming Languages” – Chapter 13

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
- If the operation is not permitted, the S.M. throws *SecurityException*
  - in some cases, this might be dangerous…

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            */
```

As an excerpt from "Concepts in Programming Languages" – Chapter 13

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