Konzepte objektorientierter Programmierung
– Lecture 10 –

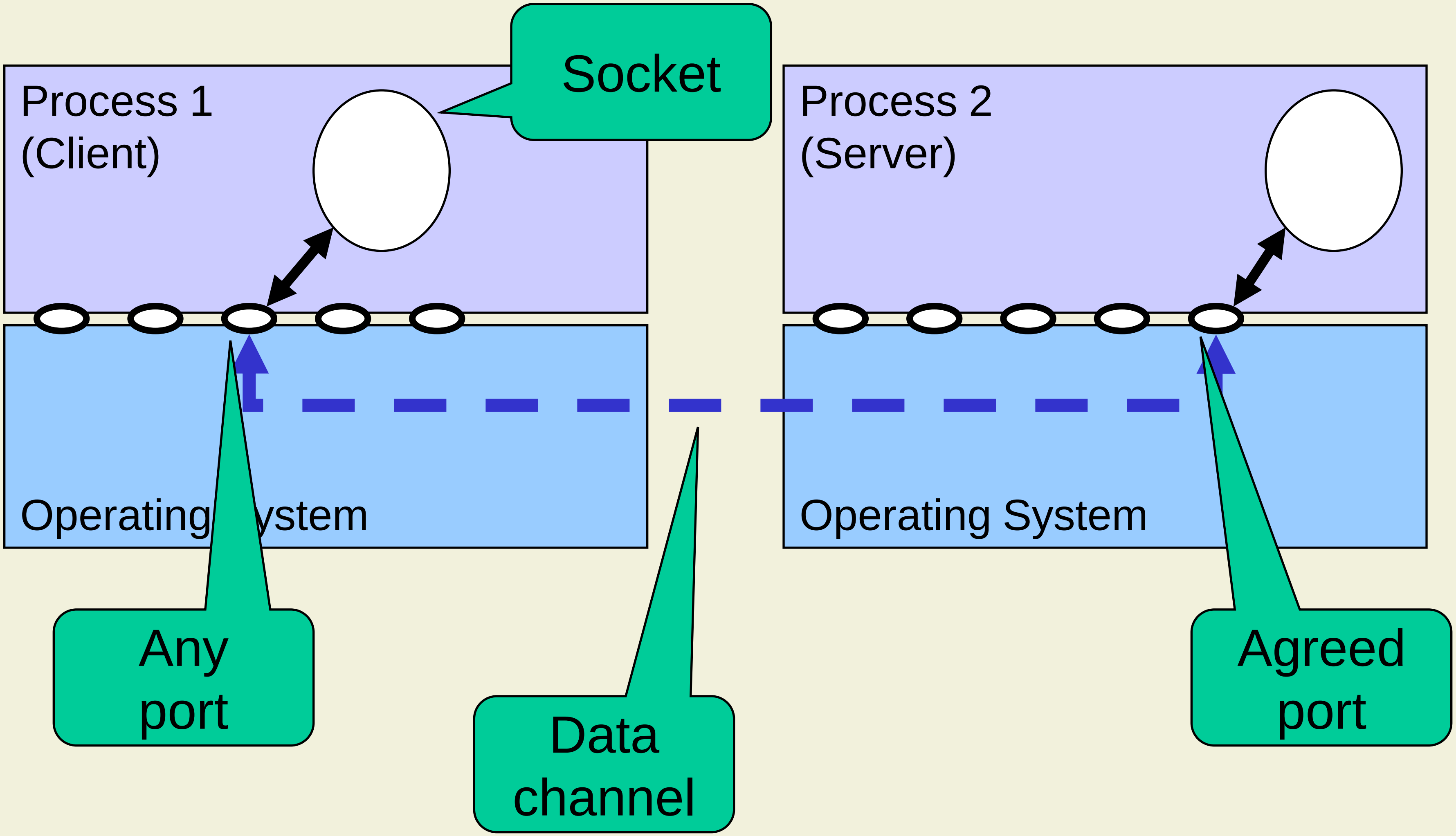
Prof. Dr. Peter Müller

Chair of Programming Methodology

Herbstsemester 2008

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Sockets and Ports



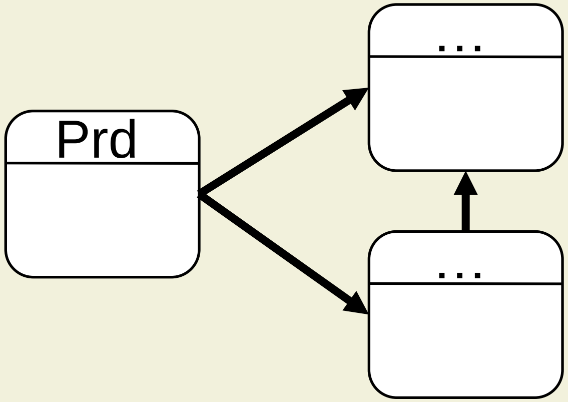
Object Streams in Java

- Serialization needs access to private fields
  - Interface Serializable is used as tag
- Object streams serialize
  - Values of primitive types
  - Serializable objects
- All objects except strings are written only once

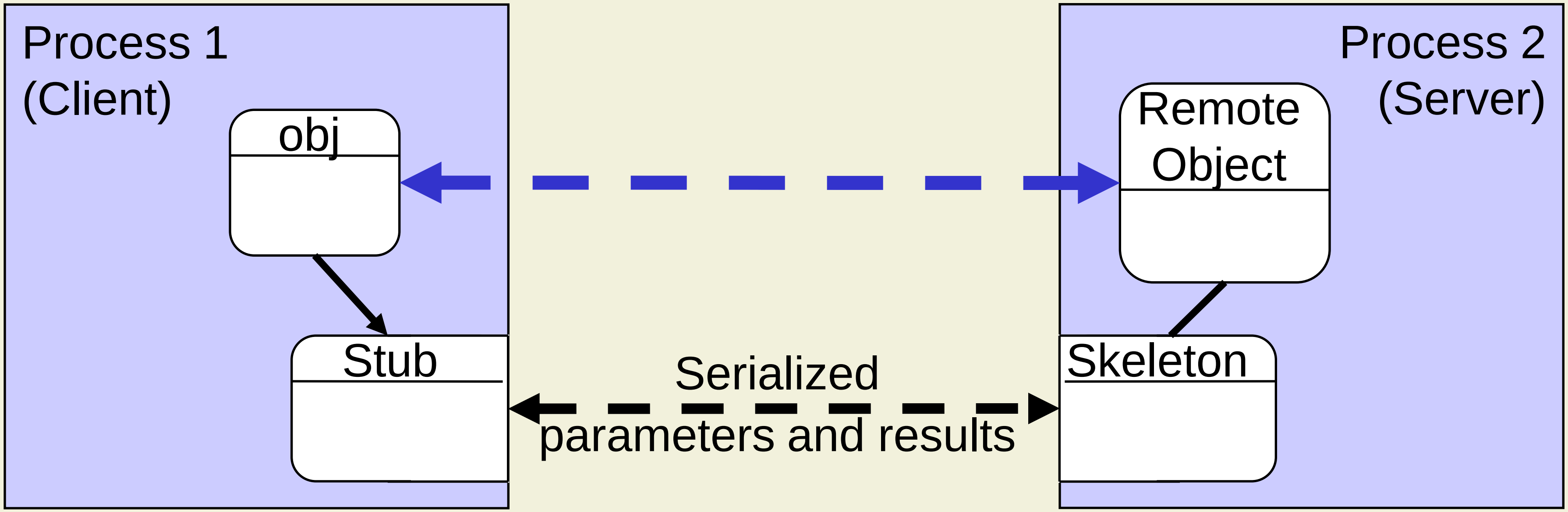
```
interface Serializable { }
```

```
class ObjectOutputStream
        extends OutputStream
        implements … {

  void writeObject( Object obj )
      throws IOException { … }
  … }
```



Stubs and Skeletons



- Remote objects are represented locally by stubs
- Stubs and skeletons provide communication
- Code for stubs and skeletons can be generated automatically (RMI compiler rmic)

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Remote Method Invocation

- Remote interfaces can be used to invoke methods of remote objects
- Communication is transparent except for
  - Error handling
  - Problems of serialization
- Coding is almost identical to local solutions

```
class Producer extends Thread {
  Buffer buf;

  Producer( Buffer b ) { buf = b; }

  void run( ) {
   while ( true )
     try {
       buf.put( new Prd( ) );
     } catch( Exception e ) { … }
  }
}
```

Agenda for Today

10. Mobile Code

10.1 Reflection

10.2 Dynamic Class Loading

10.3 Bytecode Verification

Objectives

- Mobile code
- Security and type safety

10. Mobile Code

10.1 Reflection

10.2 Dynamic Class Loading

10.3 Security

Repetition: Dynamic Type Checking

- **instanceof** can be used to avoid runtime errors
- **instanceof** makes type information available to programs

```
Object[ ] oa = new Object[ 10 ];
String s = "A String";

oa[ 0 ] = s;

…
if ( oa[ 0 ] instanceof String )
 s = (String) oa[ 0 ];

s = s.concat( "Another String" );
```

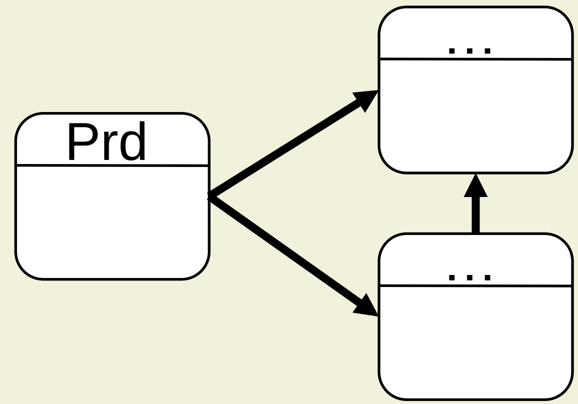
Object Streams in Java

- Serialization needs access to private fields
  - Interface Serializable is used as tag
- Object streams serialize
  - Values of primitive types
  - Serializable objects
- All objects except strings are written only once

```
interface Serializable { }
```

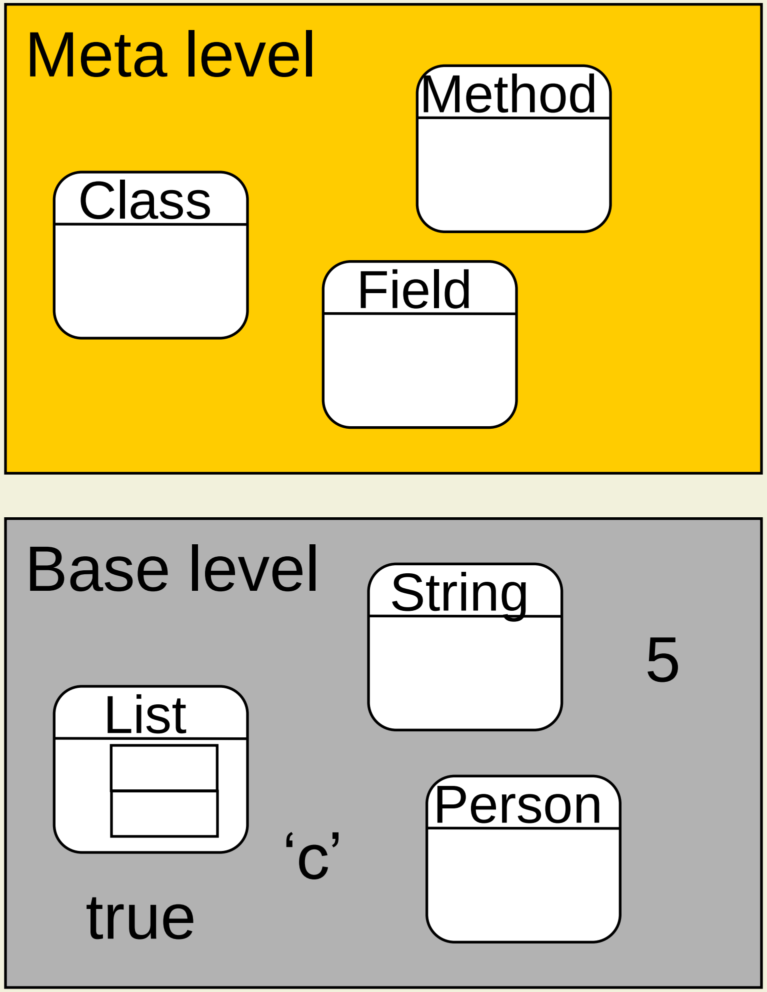
```
class ObjectOutputStream
        extends OutputStream
        implements … {

  void writeObject( Object obj )
      throws IOException { … }
  … }
```



Reflection

- Runtime meta information
  - Data about structure and properties of base data
- Simplest form
  - Type information is available at runtime
- Most elaborate
  - All compile time information is available at runtime
  - Examples: Methods of a class, parameter and result types of methods, etc.



Class Objects

```
class Class … {
  static  Class      forName( String name ) throws …                {…}
          Method[ ]  getMethods( )                                   {…}
          Method[ ]  getDeclaredMethods( )                           {…}
          Method     getMethod( String name, Class[ ] parTypes )     {…}
          Class      getSuperclass( )                                {…}
          boolean    isAssignableFrom( Class cls )                   {…}
          Object     newInstance( ) throws …                         {…}
  … }
```

- The Class object for a class can be obtained by the pre-defined class-field

```
Class StringClass = String.class;
```

Example: Inspection

```
import java.lang.reflect.*;

public class FieldInspector {
  public static void main( String[ ] ss ) {
    Class cl = Class.forName( ss[ 0 ] );
    Field[ ] fields = cl.getFields( );
    for( int i = 0; i < fields.length; i++ ) {
      Field f = fields[ i ];
      Class type = f.getType( );
      String name = f.getName( );
      System.out.println( type.getName( ) + " “ + name + “;“ );
    }
  }
}
```

Error handling omitted

Example: Methods as Parameters

Static method with signature Boolean (String)

```
static void apply( Method filter, String[ ] s ) throws Exception {
  Object[ ] par = new Object[ 1 ];
  for ( int i=0; i < s.length; i++ ) {
    par[ 0 ] = s[ i ];
    if ( ( ( Boolean ) filter.invoke( null, par ) ).booleanValue( ) )
      System.out.println( s[ i ] );
  }
}
```

Typecast necessary

Target object

Parameter array

Methods as Parameters (cont'd)

```
import java.lang.reflect.*;

class Filter {
  static void apply( Method filter, String[ ] s ) throws Exception { … }
  static Boolean single( String s ) { … }
  static Boolean startA( String s ) { … }
  static void main( String[ ] args ) throws Exception {
    Class[ ] parTypes = { String.class };
    if ( args[ 0 ].equals( "1“ ) )
      apply( Filter.class.getMethod( "single",parTypes ), args );
    else if ( args[ 0 ].equals( “2“ ) )
      apply( Filter.class.getMethod( "startA",parTypes ), args );
  }
}
```

Problems

```
public Object invoke( Object obj, Object[ ] args )
    throws IllegalAccessException,
           IllegalArgumentException,
           InvocationTargetException { … }
```

- **Safety checks** have to be done **at runtime**
  - Syntax checking (number of arguments)
  - Type checking (of arguments and results)
  - Accessibility (of fields and methods)
- Exceptions of underlying methods are caught and wrapped

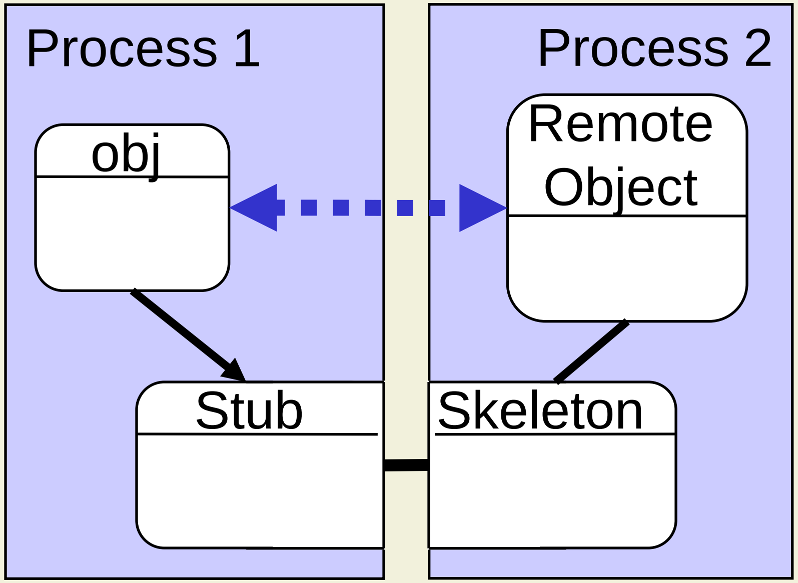
Applications of Reflection

- Serialization
- Persistence (e.g., Java Beans)
- Passing methods as arguments
- Some design patterns (e.g., visitor)
- Debugging
- Dynamic class Loading

10. Mobile Code

Motivation

- Distributed programs require code to be available on all machines
- Difficult to deploy and to maintain
- Dynamic deployment necessary for, e.g., web programming (applets)
- Solution: Code on demand

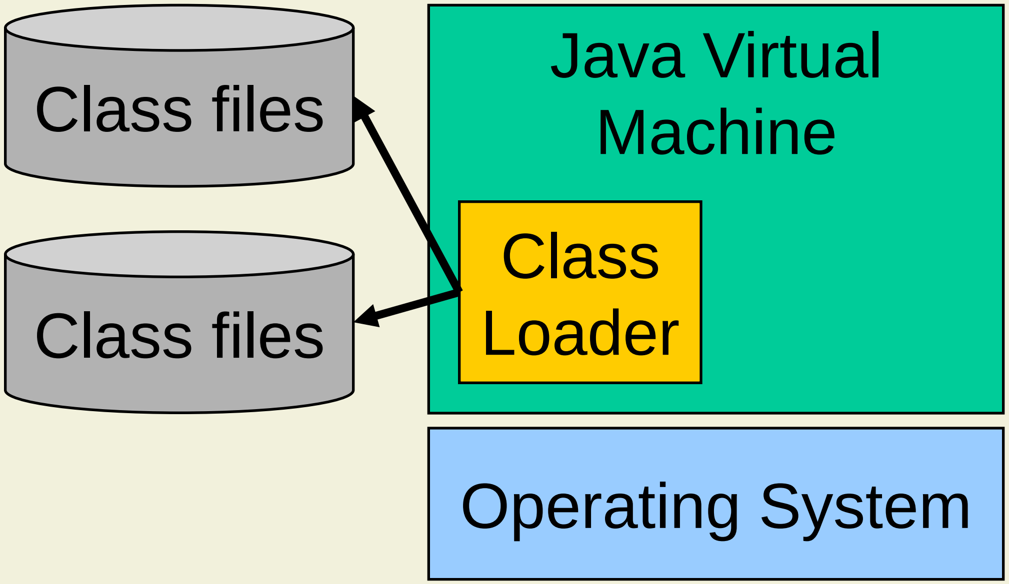


Process 1:
- Remote interfaces
- Parameter and result types
- Stubs
- Etc.

Process 2:
- Remote interfaces
- Parameter and result types
- Skeletons
- Etc.

Class Loaders

- Programs are compiled to bytecode
  - Platform-independent format
  - Organized into class files
- Bytecode is interpreted on a virtual machine
- Class loader gets code for classes and interfaces on demand
- Programs can contain their own class loaders



Example: Specialized Class Loader

Error handling partly omitted

```
public class MyLoader extends ClassLoader {
  byte[ ] getClassData( String name ) { … }

  public synchronized Class loadClass( String name )
      throws ClassNotFoundException {

    Class c = findLoadedClass( name );
    if ( c!=null ) return c;

    try { c = findSystemClass( name );  return c; }
    catch ( ClassNotFoundException e ) { }

    byte[ ] data = getClassData( name );
    return defineClass( name, data, 0, data.length );  }
}
```

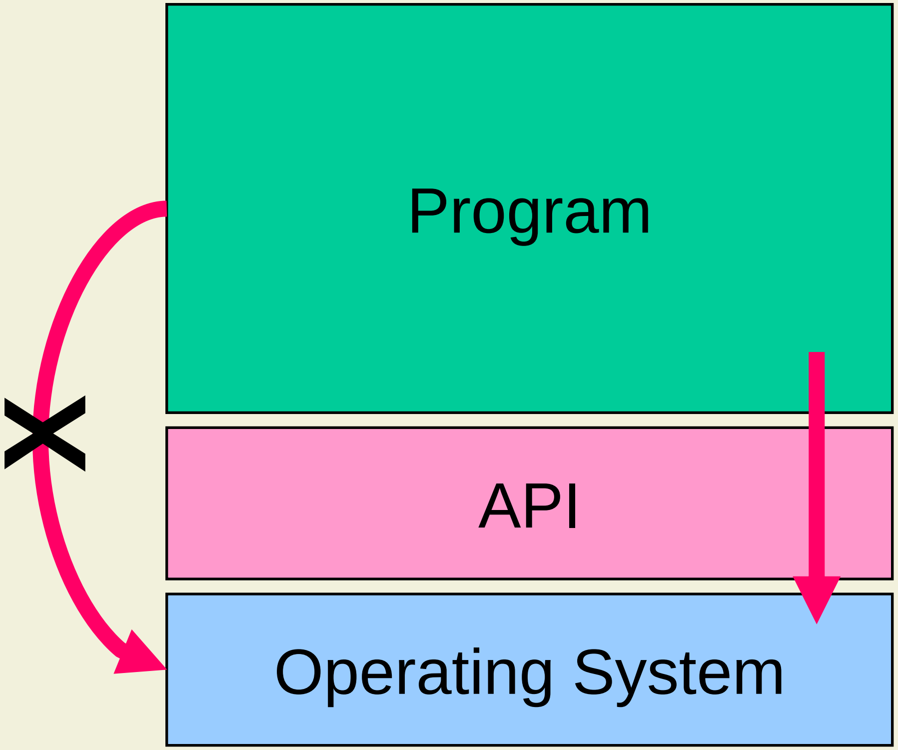
10. Mobile Code

Security in Mobile Environments

- Mobile code enables
  - Download and execution of code, e.g., Java applets
  - Upload of code, e.g., to customize servers
- Security issue: **Mobile code cannot be trusted**
  - Code may not be type safe
  - Code may destroy or modify data
  - Code may expose personal information
  - Code may crash the underlying VM
  - Code may purposefully degrade performance (denial of service)

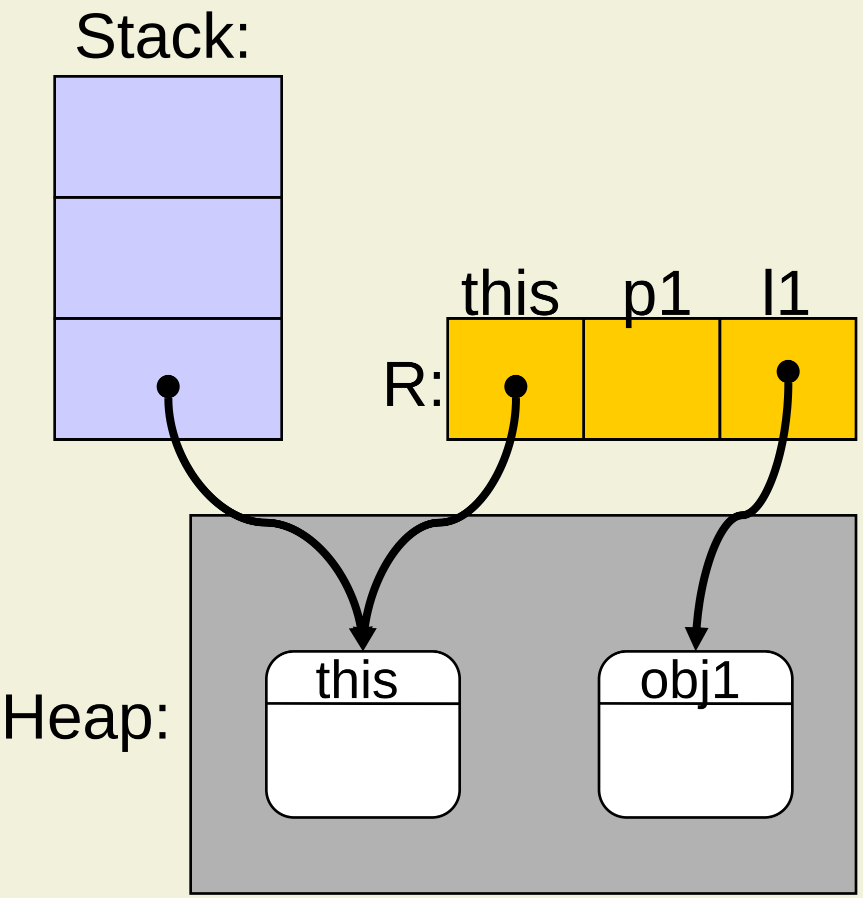
Security for Java Programs

- Sandbox
  - Applets get access to system resources only through an API
  - Access control can be implemented
- Security relies on
  - Type safety
  - Code does not by-pass sandbox



Java Virtual Machine

- JVM is stack-based
- Most operations pop operands from a stack and push a result
- Registers store method parameters and local variables
- Stack and registers are part of the method activation record

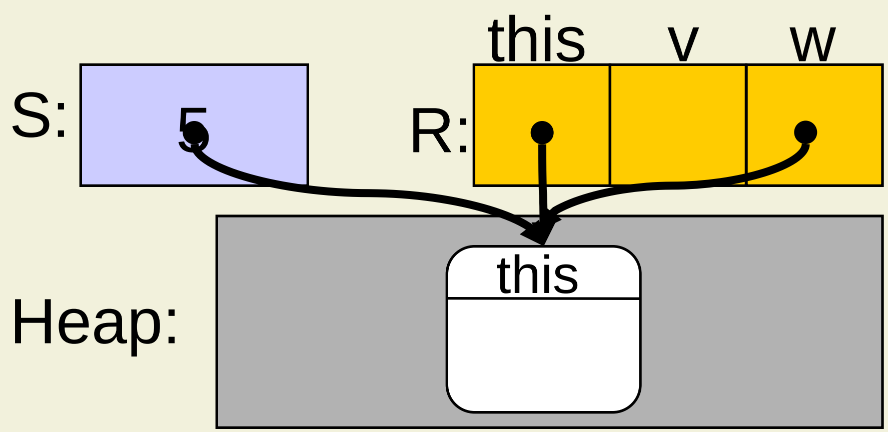


Java Bytecode

- Instructions are typed
- Load and store instructions access registers
- Control is handled by intra-method branches (goto, conditional branches)

```
class C {
  void m( ) {
    int v;
    Object w;
    v = 5;
    w = this;
  }
}
```

```
iconst 5
istore 1
aload 0
astore 2
return
```



Bytecode Verification

- Proper execution requires that
  - Each instruction is type safe
  - Only initialized variables are read
  - No stack over- or underflow occurs
  - Etc.

- Java Virtual Machine guarantees these properties
  - By **bytecode verification** when a class is loaded
  - By **dynamic checks at runtime**

Bytecode Verification via Type Inference

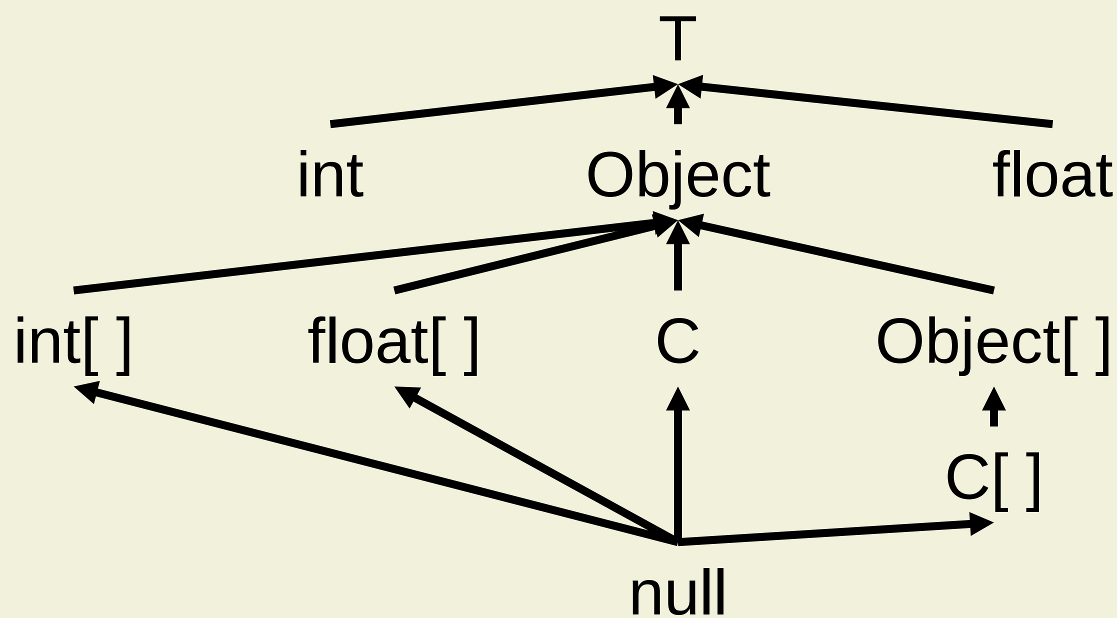
- The Bytecode verifier **simulates** the execution of the program
- Operations are performed on **types instead of values**
- For each instruction, a rule describes how the **stack and local variables** are modified

$$\textbf{i:}\ (S,R) \rightarrow (S',R')$$
$$\textbf{iadd}:\ (int.int.S,R) \rightarrow (int.S,R)$$

- Errors are denoted by the **absence of a transition**
  - Type mismatch
  - Stack over- or underflow

Types of the Inference Engine

- Primitive types
- Object and array reference types
- null type for the null reference
- T for uninitialized registers



Selected Rules

- Maximum stack size (MS) and maximum number of parameters and local variables (ML) are stored in the classfile
- Rule for method invocation uses method signature (no jump)

iconst n:

$(S,R) \rightarrow (int.S,R)$, if $|S| < MS$

iload n:

$(S,R) \rightarrow (int.S,R)$,
if $0 \leq n \leq ML \wedge R(n) = int \wedge |S| < MS$

astore n:

$(t.S,R) \rightarrow (S,R\{ n \leftarrow t \})$,
if $0 \leq n \leq ML \wedge t <: Object$

invokevirtual C.m.σ:
$(t'_n \ldots t'_1.t'.S,R) \rightarrow (r.S,R)$, if
$\sigma = r (t_1,\ldots,t_n) \wedge t' <: C \wedge t'_i <: t_i$

Example

int v;	**iconst** 5	([ ] , [C,T,T]) →
Object w;	**istore** 1	(int , [C,T,T]) →
v = 5;	**aload** 0	([ ] , [C,int,T]) →
w = **this**;	**astore** 2	(C , [C,int,T]) →
	return	([ ] , [C,int,C])

this — v — w

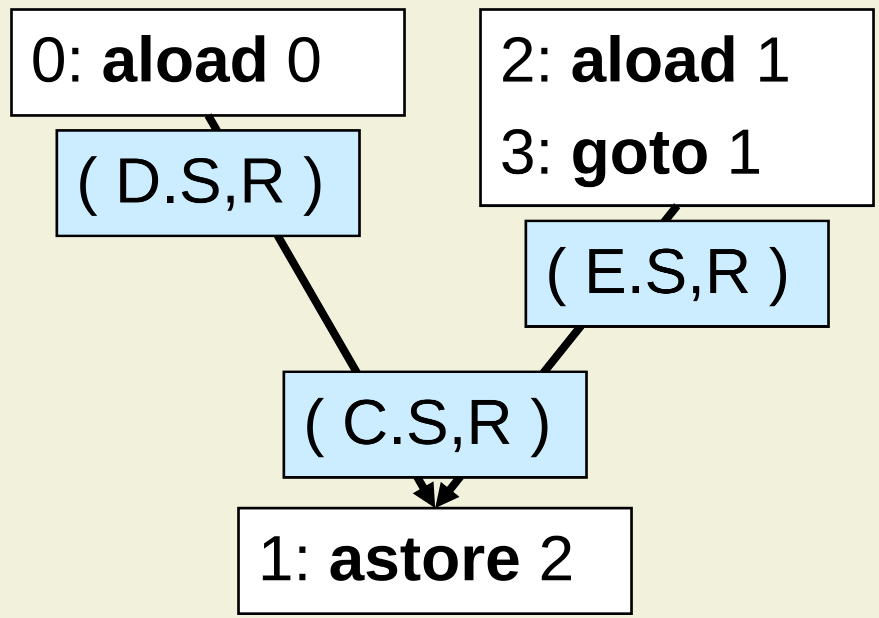
int v;	**iconst** 5	([ ] , [C,T,T]) →
Object w;	**istore** 1	(int , [C,T,T]) →
v = 5;	**iload** 1	([ ] , [C,int,T]) →
w = v;	**astore** 2	(int , [C,int,T])
	return	**stuck**

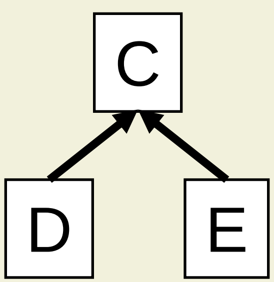
astore expects an object type on top of the stack!

Smallest Common Supertype

- Branches lead to **joins in control flow**
- Instructions can have **several predecessors**
- **Smallest common supertype** is selected (T if no other common supertype exists)

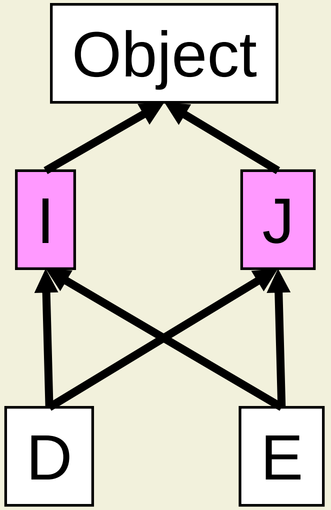
```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```





Handling Multiple Subtyping

- With multiple subtyping, **several smallest common supertypes** may exist
- JVM solution
  - Ignore interfaces
  - Treat all interface types as Object
  - Works because of single inheritance of classes
- Problem
  - **invokeinterface** I.m cannot check whether target object implements I
  - Runtime check is necessary



Inference Algorithm

- Inference is a fixpoint iteration

```
in( 0 ) := ( [ ] , [ P_0,…,P_n,T,…,T ] )
worklist := { i | instr_i is an instruction of the method }
while worklist ≠ ∅ do
  i := min( worklist )
  remove i from worklist
  out( i ) := apply_rule( instr_i, in( i ) )
  forall q in successors( i ) do
    in( q ) := pointwise_scs( in( q ),out( i ) )
    if in( q ) has changed then worklist := worklist ∪ { q }
  end
end
```

Pointwise SCS

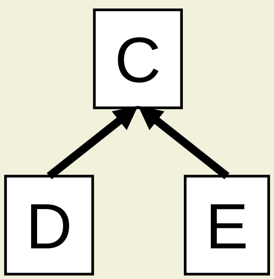
- scs(s,t) is the smallest common supertype of s and t

$$\text{pointwise\_scs}(\ ([\,s_1.\ldots.s_k\,],[\,t_0,\ldots,t_n\,]),\ ([\,s'_1.\ldots.s'_k\,],[\,t'_0,\ldots,t'_n\,])\ ) =$$
$$([\,\text{scs}(s_1,s'_1).\ldots.\text{scs}(s_k,s'_k)\,],[\,\text{scs}(t_1,t'_1),\ldots,\text{scs}(t_n,t'_n)\,])$$

- pointwise_scs is undefined for stacks of different heights
  - Bytecode verification results in an error

Inference Example

0: **aload** 0
1: **astore** 2
2: **aload** 1
3: **goto** 1



	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T]) ([C] , [D,E,T]) ([C] , [D,E,T])	([ ] , [D,E,D]) ([ ] , [D,E,C])
2:	([ ] , [D,E,D]) ([ ] , [D,E,C])	([E] , [D,E,D]) ([E] , [D,E,C])
3:	([E] , [D,E,D]) ([E] , [D,E,C])	([E] , [D,E,D]) ([E] , [D,E,C])

worklist

0 1 2 3

Type Inference: Discussion

- Advantages
  - Determines the most general solution that satisfies the typing rules
  - Might be more general than what is permitted by compiler
  - Very little type information required in class file
- Disadvantages
  - Fixpoint computations may be slow
  - Solution for interfaces is imprecise and requires runtime checks
- Alternative: type checking (since Java 6)

Bytecode Verification via Type Checking

- Extend class file to store type information

  (int , [C,int,T])

- Type information can be declared for each bytecode instruction
- Type information required at the beginning of all basic blocks:
  - At jump target
  - At entry point of exception handler

  Includes all join points
- Computation of SCS no longer necessary
  - Avoid fixpoint computation and interface problem

Type Checking Algorithm

- Use and check declared types wherever available
- Infer types otherwise

foreach basic block of a method body **do**
  in := types(start)
  foreach { i | $instr_i$ is an instruction of basic block } **do**
    in := apply_rule($instr_i$, in)
    forall q in successors(i) **do**
      if types(q) is declared **then**
        check that in is assignable to types(q)
        in := types(q)
      end
  end
end

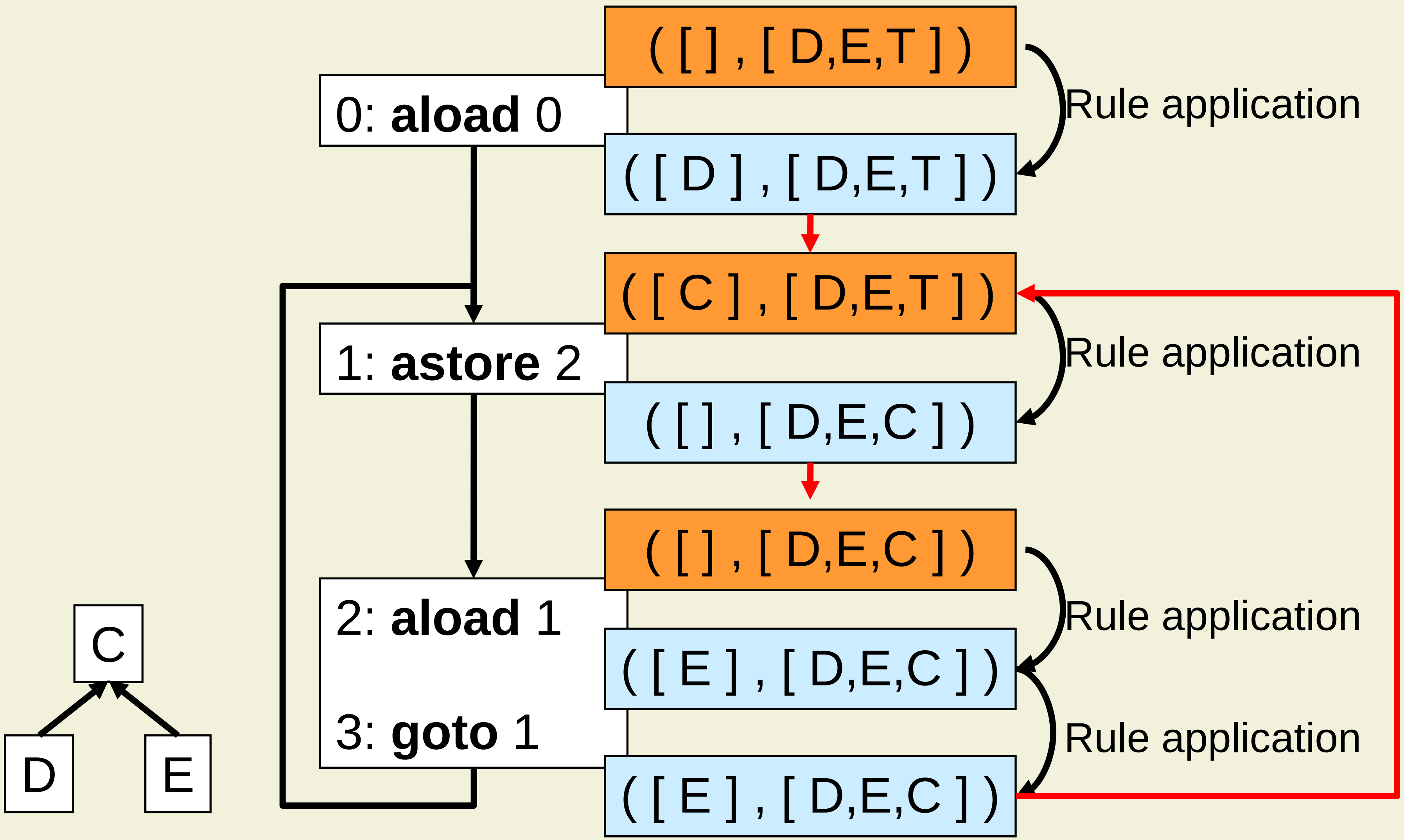
Required types

Check conditions and infer next configuration

Check declared types

Use declared types if $instr_i$ is not a jump

Type Checking Example



Summary

- Bytecode verification enables secure mobile code
  - For programs written in typed bytecode
- Bytecode verification can be done via type inference or type checking

- Suggested reading
  - Xavier Leroy.
    Java bytecode verification: algorithms and formalizations.
    Journal of Automated Reasoning 30(3-4):235-269, 2003.
    Available from the course web site