

# Concepts of Object-Oriented Programming

**Peter Müller**

Chair of Programming Methodology

Autumn Semester 2021

**ETH** zürich

# Meeting the Requirements

## Cooperating Program Parts with Well-Defined Interfaces

- Objects (data + code)
- Interfaces
- Encapsulation

## Classification and Specialization

- Classification, subtyping
- Polymorphism
- Substitution principle

## Highly Dynamic Execution Model

- Active objects
- Message passing

## Correctness

- Interfaces
- Encapsulation
- Simple, powerful concepts

# Topics in this Section

- Cooperating program parts ...
  - How do we define components?
  
- ... with well-defined interfaces
  - What is the interface of a component?
  - How do we describe the interface of a component?
  - How do we make sure clients use a component correctly?



Types  
and  
contracts

# 4. Types

## 4.1 Bytecode Verification

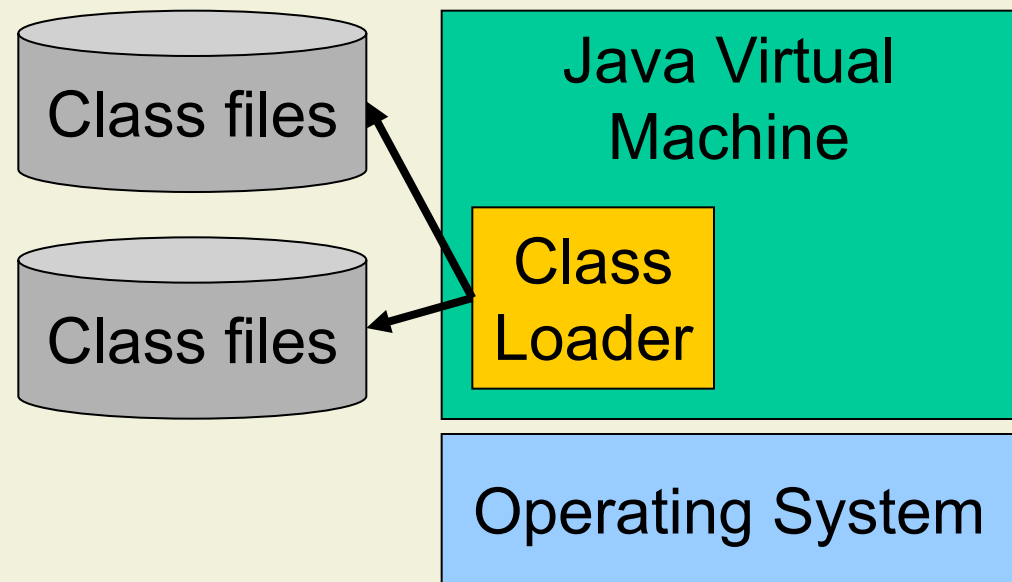
## 4.2 Parametric Polymorphism

# Mobile Code: Motivation

- Download and execution of code, e.g., Java applets
  - Web pages
  - Mobile devices
  
- Upload of code
  - Customizing servers
  
- Automatic distribution of code and patches in distributed systems

# 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 ); }  
}
```

Java

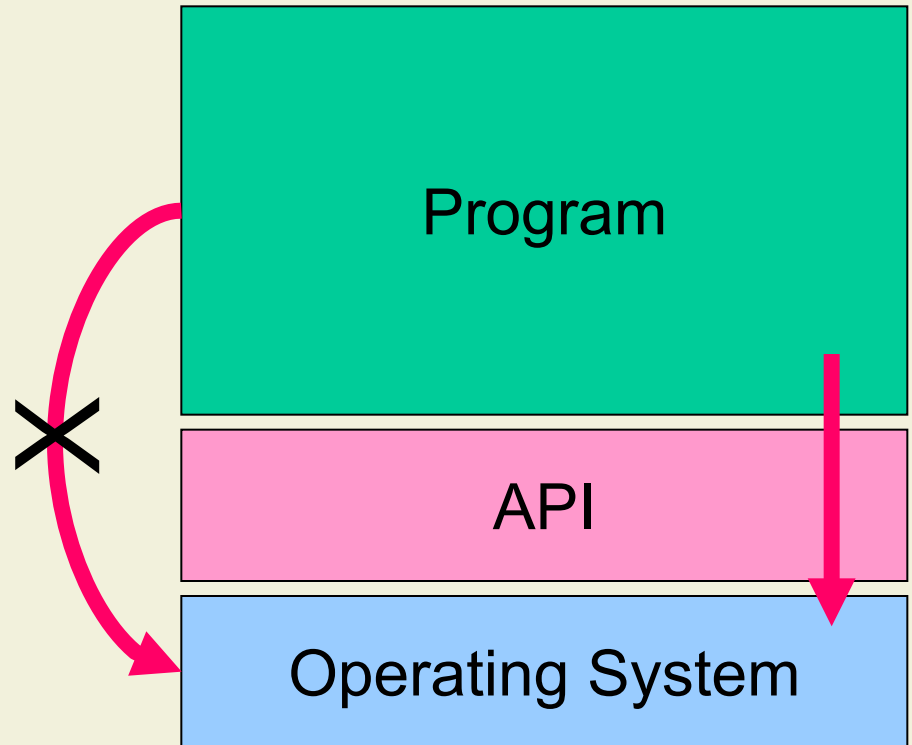
# Security for Java Programs

## ■ 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



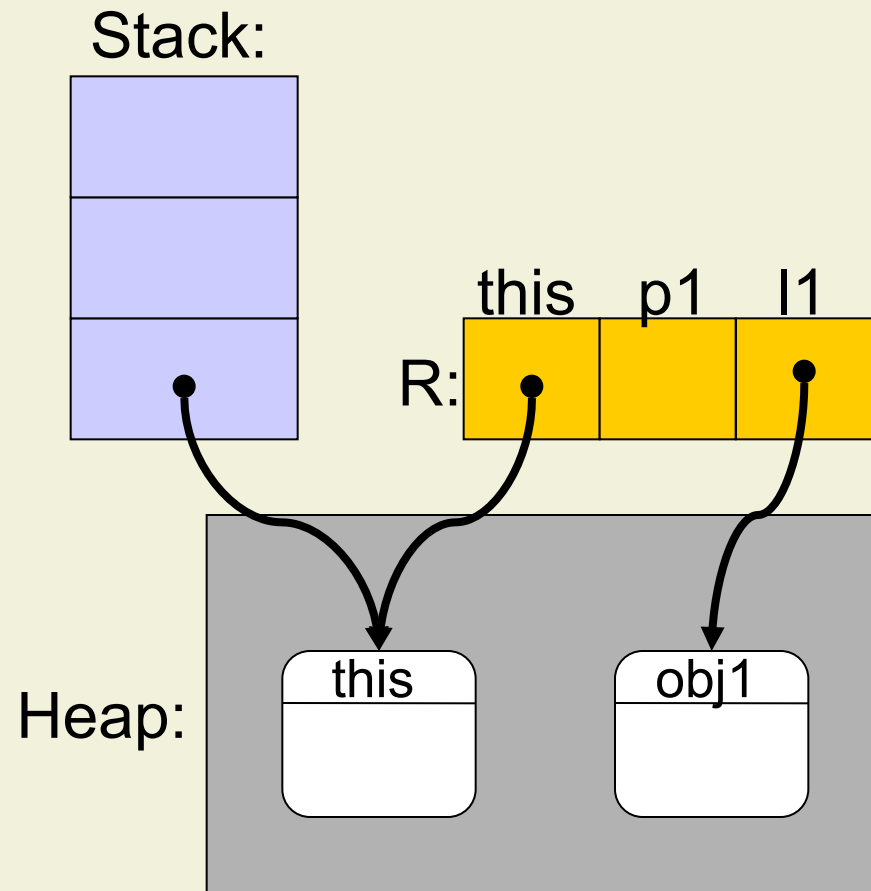


# Security in Mobile Environments

- 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)
  
- How to guarantee a minimum level of security?
  - Untrusted code producer
  - Untrusted compiler

# 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

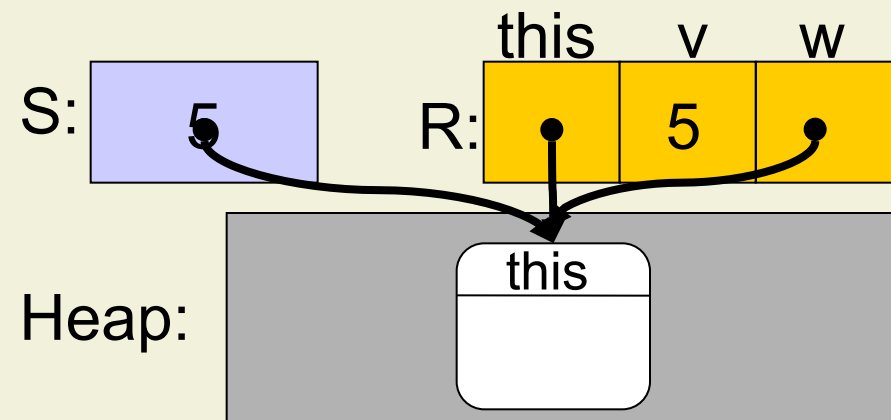


# Java Bytecode

- Instructions are typed
- Load and store instructions access registers
- Control is handled by intra-method jumps (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 correct
  - 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 run time**

# Bytecode Verification via Type Inference

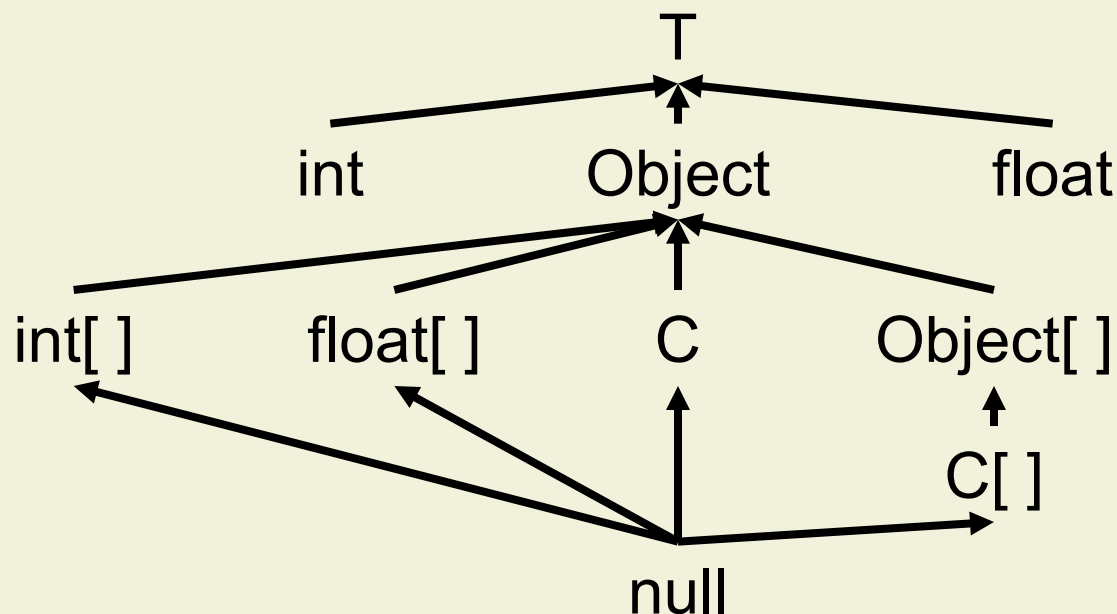
- The Bytecode verifier **simulates** the execution of the program
- Operations are performed on **types instead of values**
- For each instruction, a rule describes how the **operand stack and local variables** are modified

$$\begin{aligned} i: (S, R) &\rightarrow (S', R') \\ \text{iadd}: (\text{int.int.S}, R) &\rightarrow (\text{int.S}, R) \end{aligned}$$

- 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 (MR) are stored in the classfile
- Rule for method invocation uses method signature (no jump)

iconst n:

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

iload n:

$(S, R) \rightarrow (\text{int}.S, R),$   
 $\text{if } 0 \leq n < \text{MR} \wedge R(n) = \text{int} \wedge |S| < \text{MS}$

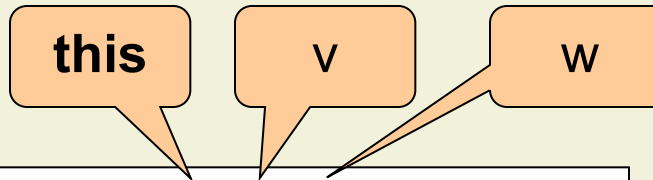
astore n:

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

invokevirtual C.m. $\sigma$ :

$(t'_n \dots t'_1.t'.S, R) \rightarrow (r.S, R), \text{ if}$   
 $\sigma = r(t_1, \dots, t_n) \wedge t' <: C \wedge t'_i <: t_i$

# Example



```
int v;
Object w;
v = 5;
w = this;
```

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

```
( [ ] , [ C,T,T ] ) →
( int , [ C,T,T ] ) →
( [ ] , [ C,int,T ] ) →
( C , [ C,int,T ] ) →
( [ ] , [ C,int,C ] )
```

```
int v;
Object w;
v = 5;
w = v;
```

```
iconst 5
istore 1
iload 1
astore 2
return
```

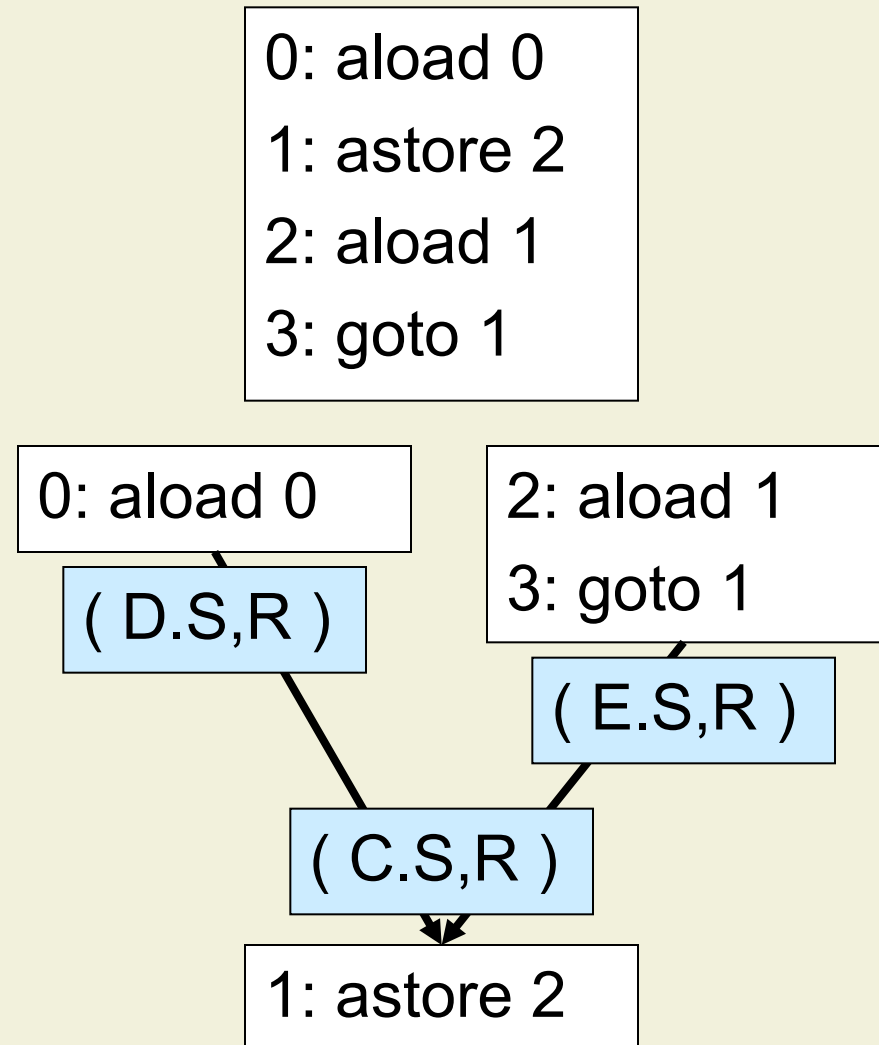
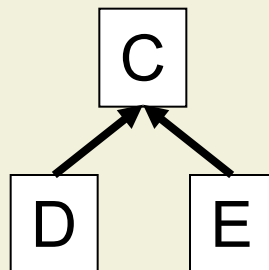
```
( [ ] , [ C,T,T ] ) →
( int , [ C,T,T ] ) →
( [ ] , [ C,int,T ] ) →
( int , [ C,int,T ] )
stuck
```

astore  
expects an  
object type  
on top of  
the stack!



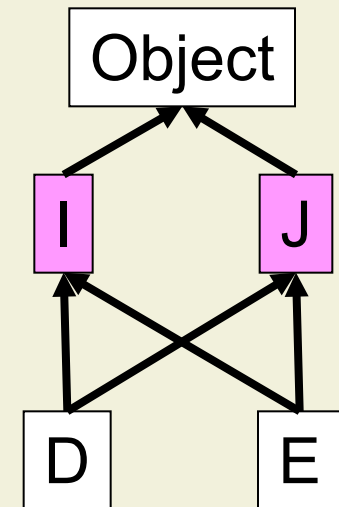
# Smallest Common Supertype

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



# 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
  - Run-time check is necessary



# Inference Algorithm

- Inference is a fixpoint iteration

```
in( 0 ) := ( [ ] , [ P0, ..., Pn, T, ..., T ] )  
worklist := { i | instri is an instruction of the method }  
while worklist ≠ ∅ do  
  i := min( worklist )  
  remove i from worklist  
  out( i ) := apply_rule( instri, in( i ) )  
  foreach 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

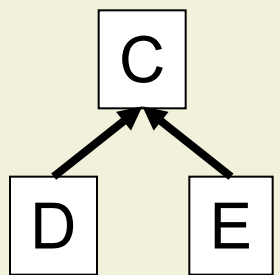
- $\text{scs}(s, t)$  is the smallest common supertype of  $s$  and  $t$

$$\begin{aligned} \text{pointwise\_scs} \big( & ([s_1, \dots, s_k], [t_0, \dots, t_n]), \\ & ([s'_1, \dots, s'_k], [t'_0, \dots, t'_n]) \big) = \\ & ([\text{scs}(s_1, s'_1), \dots, \text{scs}(s_k, s'_k)], [\text{scs}(t_0, t'_0), \dots, \text{scs}(t_n, t'_n)]) \end{aligned}$$

- $\text{pointwise\_scs}$  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



worklist

0 1 2 3

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

# 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 run-time 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 )

Required  
types

**foreach** { i | instr<sub>i</sub> is an instruction of basic block } **do**

    in := apply\_rule( instr<sub>i</sub>, in )

Check conditions and infer  
next configuration

**foreach** q in successors( i ) **do**

**if** types( q ) is declared **then**

        check that in is assignable to types( q )

**end**

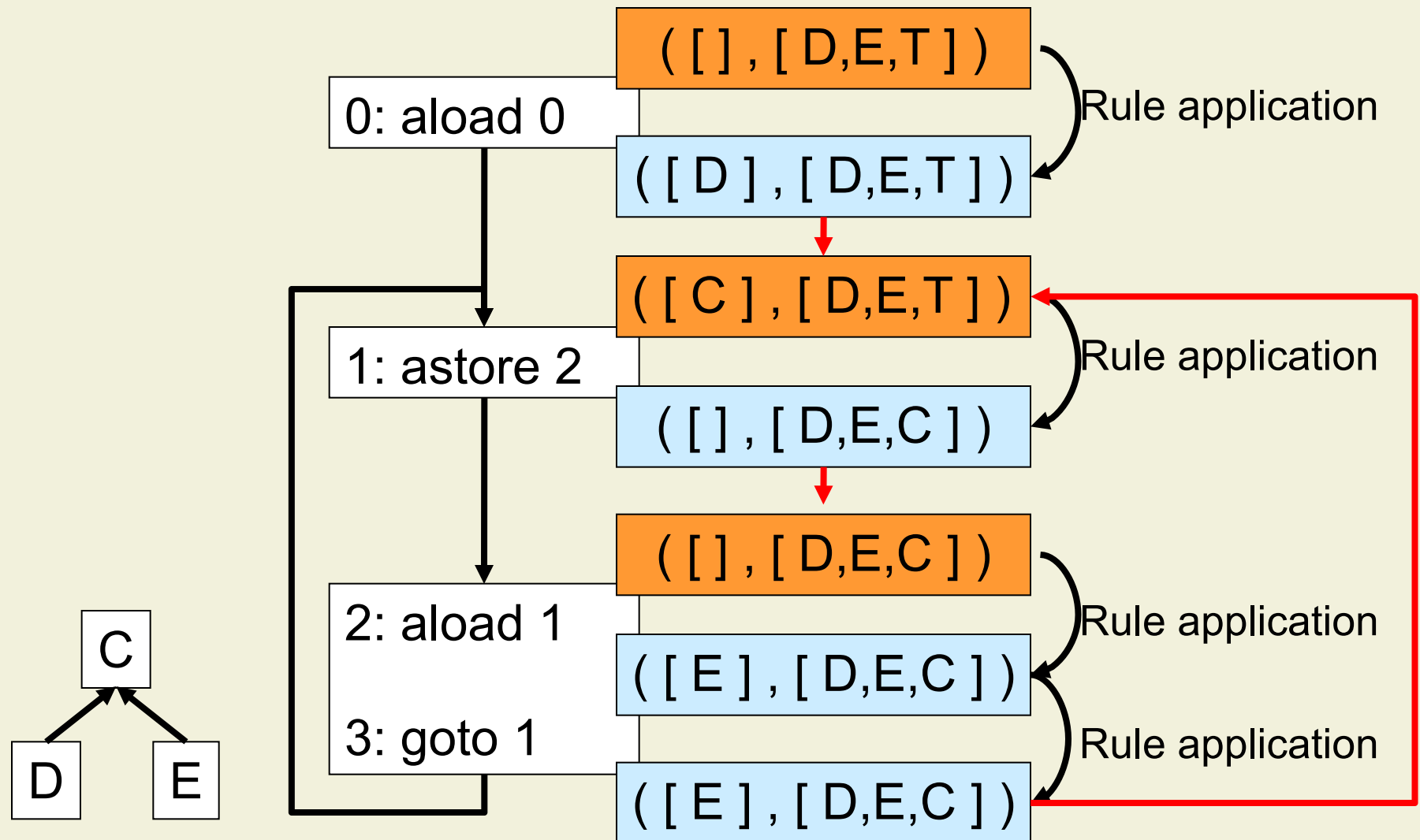
**end**

**end**

Check  
declared types



# Type Checking Example



# Bytecode Verification: Summary

- Bytecode verification enables secure mobile code
  - For programs written in typed bytecode
- Bytecode verification can be done via **type inference** or **type checking**
- Some run-time type checks are still necessary
  - For instance, casts and co-variant arrays

# Type Inference for Source Programs

- Type inference can also be done on source code
  - For example, C# 3.0 and Scala **infer types of local variables**
  - **Reduce annotation overhead**, especially with generics
- Type annotations can still be used to support inference

```
def sum( a: Array[ Int ] ): Int = {  
  val it = a.elements  
  var s = 0;  
  while( it.hasNext ) { s = s + it.next }  
  s  
}
```

Scala

```
def client = {  
  var a = 1  
  a = "Hello"  
}
```

Scala

```
def client = {  
  var a: Any = 1  
  a = "Hello"  
}
```

Scala

# Type Inference vs. Dynamic Typing

- Type inference determines the static type automatically and then performs static type checking
- Dynamic typing does not require a static type and does not perform static type checking

```
void client( ) {  
    var a = 1;  
    a = "Hello";  
}
```

C#

```
void client( ) {  
    dynamic a = 1;  
    a = "Hello";  
}
```

C#

# Inference of Method and Field Types

- Inference of method signatures generally requires knowledge of all implementations
- Inference of field types generally requires knowledge of all accesses to the field
- Inference of these types is non-modular
  - Or based on speculation

```
class A {  
  var f = 5;  
  def foo( p: Int ) = {  
    p  
  }  
}
```

Inference: f: Int

Inference: foo returns Int

Scala

```
class B extends A {  
  f = "Hello";  
  override def foo( p: Int ) = {  
    "Hello"  
  }  
}
```

Scala

# 4. Types

4.1 Bytecode Verification

4.2 Parametric Polymorphism

# Polymorphism Revisited

- Not all polymorphic code is best expressed using subtype polymorphism
- Recovering precise type information requires **downcasts**
- Subtype relations are sometimes **not desirable**
  - E.g., covariant arrays

```
class Queue {  
    Object elem;  
    Queue next;  
    void enqueue( Object e ) { ... }  
    Object dequeue( ) { ... }  
}
```

Java

```
Queue q = new Queue( );  
String s = "Hello";  
q.enqueue( s );  
String t = ( String ) q.dequeue( );
```

Java

```
static void fill( Object[ ] a, Object val )  
{ ... }
```

Java

# Parametric Polymorphism

- Classes and methods can be **parameterized with types**
- Clients provide instantiations for type parameters
- **Modularity**: generic code is type checked once and for all (without knowing the instantiations)

```
class Queue<T> {  
    T elem;  
    Queue<T> next;  
    void enqueue( T e ) { ... }  
    T dequeue( ) { ... }  
}
```

Java

```
Queue<String> q;  
q = new Queue<String>( );  
String s = "Hello";  
q.enqueue( s );  
String t = q.dequeue( );
```

Java

```
static <T> void fill( T[ ] a, T val )  
{ ... }
```

Java



# Type Checking Generic Code

- Type checking a generic class often **requires information about its type arguments**
  - Availability of methods
- Constraints can be expressed by specifying **upper bounds** on type parameters

```
class Queue<T> {  
    T elem;  
    Queue<T> next;  
  
    void enqueue( T e ) {  
        if( next == null ) { ... }  
        else {  
            if( e.compareTo( elem ) <= 0 ) {  
                next.enqueue( elem );  
                elem = e;  
            } else next.enqueue( e );  
        }  
    }  
    ...  
}
```

Java

# Upper Bounds: Example

```
interface Comparable<T> {
    int compareTo( T o );
}
```

Java

```
Queue<String> q;
// String implements
// Comparable<String>
```

Java

```
Queue<Person> q;
// Person does not
// implement
// Comparable<Person>
```

Java

```
class Queue<T extends Comparable<T>> {
    T elem;
    Queue<T> next;

    void enqueue( T e ) {
        if( next == null ) { ...
        else {
            if( e.compareTo( elem ) <= 0 ) {
                next.enqueue( elem );
                elem = e;
            } else next.enqueue( e );
        }
    }
    ...
}
```

Typecheck under the  
assumption  
 $T \leq \text{Comparable}\langle T \rangle$

Java

# Subtyping and Generics

```
class Queue<T extends Comparable<T>> { ... }
```

- Generic types are subtypes of their declared supertypes
- Type variables are subtypes of their upper bounds
- How about different instantiations of the same generic class?

```
Object o = new Queue<String>( );
```

```
void foo( T p ) {  
    Comparable<T> v = p;  
}
```

```
List<Person> o;  
o = new List<Student>( );  
o = new List<Object>( );
```

# Covariant Type Arguments

- Covariance:  
If  $S \leq T$  then  
 $C<S> \leq C<T>$
- Covariance is unsafe when a generic type argument is used for variables that are written by clients
  - Mutable fields
  - Method arguments

```
class Queue<T> {  
    void enqueue( T e ) { ... }  
    T dequeue( ) { ... }  
}
```

```
void put( Queue<Object> q ) {  
    q.enqueue( "Hello" );  
}
```

Not type safe if q had  
type Queue<Integer>

```
Object get( Queue<Object> q ) {  
    return q.dequeue( );  
}
```

# Contravariant Type Arguments

- Contravariance:  
If  $S \leq T$  then  
 $C\langle T \rangle \leq C\langle S \rangle$
- Contravariance is unsafe when a generic type argument is used for variables that are read by clients
  - Fields
  - Method results

```
class Queue<T> {  
    void enqueue( T e ) { ... }  
    T dequeue( ) { ... }  
}
```

```
void put( Queue<String> q ) {  
    q.enqueue( "Hello" );  
}
```

```
String get( Queue<String> q ) {  
    return q.dequeue( );  
}
```

Not type safe if q had  
type Queue<Object>

# Java/C# Solution: Non-Variance

- Generic types in Java/C# are **non-variant** (neither co- nor contravariant)
- Non-variance is **statically type safe**
  - No run-time checks needed
- Non-variance is sometimes overly restrictive

```
class Queue<T> {  
    void enqueue( T e ) { ... }  
    T dequeue( ) { ... }  
}
```

Java

```
Queue<Object> o;  
o = new Queue<String>( );
```

Java

```
Queue<String> o;  
o = new Queue<Object>( );
```

Java

```
class Random<T> {  
    T next( ) { ... }  
}
```

# Java/C#: Generics vs. Arrays

- Recall: Java/C# arrays are covariant
- But an array `T[ ]` is not much different from a class `Array<T>`
- Run-time checks
  - Covariant arrays require run-time checks for each update
  - Covariant generics would need checks for field updates and argument passing
- Covariant generics would require more run-time checks in more bytecode instructions

```
Object[ ] o;  
o = new String[ 5 ];
```

Java

```
Queue<Object> o;  
o = new Queue<String>( );
```

Java

# Eiffel Solution: Covariance

- Generic types in Eiffel are **covariant**
- Design is consistent with covariance for method arguments and fields
  - But **not statically type safe**

```
class Queue[ T ] ... end
```

Eiffel

```
o: Queue[ ANY ]  
s: Queue[ STRING ]  
create s.make  
o := s
```

Eiffel

```
o: Queue[ ANY ]  
s: Queue[ STRING ]  
create o.make  
s := o
```

Eiffel



# Dart

- The type system is unsound, due to the covariance of generic types. This is a deliberate choice (and undoubtedly controversial). Experience has shown that sound type rules for generics fly in the face of programmer intuition. It is easy for tools to provide a sound type analysis if they choose, which may be useful for tasks like refactoring [Dart Language Specification]
- makes me wanna cry. It's like saying that gravity flies into the face of astronauts. Let's violate basic laws of math simply because, allegedly, they are not "intuitive"? [...]

Andreas Rossberg

# Scala Solution: Variance Annotations

- By default, generic types in Scala are **non-variant**
- Programmers can supply **variance annotations** to allow **co-** and **contravariance**
- Type checker imposes **restrictions** on use of variance annotations

```
class Queue[T] {  
  def enqueue( e: T ) = { ... }  
  def dequeue: T = { ... }  
}
```

Scala

```
Queue[ AnyRef ] o;  
o = new Queue[ String ]( );
```

Scala

```
Queue[ String ] o;  
o = new Queue[ AnyRef ]( );
```

Scala

# Covariance Annotations

- A covariance annotation (+) is useful when type variable occurs **only in positive positions**

- Result type
- Types of immutable fields

- Type checker prevents other occurrences

```
class Random[ +T ] {  
  def next: T = { ... }  
}
```

Scala

```
val r: Random[ AnyRef ] =  
  new Random[ String ]( )  
val a = r.next
```

Scala

```
class Random[ +T ] {  
  def next: T = { ... }  
  def initialize( i: T ) = { ... }  
}
```

Scala

# Contravariance Annotations

- A contravariance annotation (-) is useful when type variable occurs **only in negative positions**
  - Parameter type
- Type checker prevents other occurrences

```
class OutputChannel[ -T ] {  
  def write( x: T ) = { ... }  
}
```

Scala

```
val o: OutputChannel[ String ] =  
  new OutputChannel[ AnyRef ]( )  
o.write( "Hello" )
```

Scala

```
class OutputChannel[ -T ] {  
  def write( x: T ) = { ... }  
  def lastWritten: T = { ... }  
}
```

Scala

# Working with Non-Variant Generics

- How can we write code that works with many different instantiations of a generic class?
- Solution 1: Additional type parameters

```
static void printAll( Collection<Object> c ) {  
    for ( Object e : c ) { System.out.println( e ); }  
}
```

Java

```
foo( Collection<String> p ) {  
    printAll( p ) {  
}
```

Java

```
static <T> void printAll( Collection<T> c ) {  
    for ( T e : c ) { System.out.println( e ); }  
}
```

Java

```
foo( Collection<String> p ) {  
    printAll( p ) {  
}
```

Java

# Additional Type Parameters

```
interface Comparator<T> {  
    int compare( T fst, T snd );  
}
```

```
class TreeSet<E> {  
    TreeSet( Comparator<E> c ) { ... }  
    ... }
```

TreeSet needs to  
compare set elements

```
class Person { ... }
```

```
class Student extends Person { ... }
```

```
class PersonComp implements Comparator<Person> {  
    int compare( Person fst, Person snd ) { ... }  
}
```

Universal comparator  
for all persons

```
TreeSet<Student> s = new TreeSet<Student>( new PersonComp() );
```

Compile-time error: PersonComp is not  
a subtype of Comparator<Student>

# Additional Type Parameters (cont'd)

```
interface Comparator<T> {  
    int compare( T fst, T snd );  
}
```

```
class TreeSet<F, E extends F> {  
    TreeSet( Comparator<F> c ) { ... }  
    ... }
```

Comparator can  
compare at least Es

```
TreeSet<Person, Student> s =  
    new TreeSet<Person, Student>( new PersonComp() );
```

## ■ Drawbacks

- Additional type argument is a nuisance for clients and reveals implementation details
- Type-instantiation of Comparator cannot be changed at run time, for instance, to Comparator<Object>

## Solution 2: Wildcards

- A wildcard represents an **unknown type**

```
static <T> void printAll( Collection<T> c ) {  
    for ( T e : c ) { System.out.println( e ); }  
}
```

Java

```
static void printAll( Collection<?> c ) {  
    for ( Object e : c ) { System.out.println( e ); }  
}
```

Java



# Wildcards and Existential Types

```
static void printAll( Collection<?> c ) {  
    for ( Object e : c ) { System.out.println( e ); }  
}
```

Java

- Interpretation as **existential type**
  - “There exists a type argument T such that c has type Collection<T>”
  - Existential quantifier is instantiated automatically by the type system

```
Collection<String> c = new ArrayList<String>( );  
...  
printAll( c );
```

Java

Wildcard instantiated  
with String

# Wildcard Examples

Correct: type checker  
instantiates type argument  
with c's type argument

```
static Collection<?> id( Collection<?> c ) {  
    return c;  
}
```

Two existential  
types

```
Collection<String> c = new ArrayList<String>( );  
Collection<String> d = id( c );
```

Type error: existential  
types might have  
different instantiations  
(modular type checking)

# Wildcard Examples (cont'd)

```
static void merge( Collection<?> c, Collection<?> d ) {  
    for( Object e : c ) { d.add( e ); }  
}
```

Two existential  
types

Type error: d might  
expect elements of  
different type

# Wildcard Examples (cont'd)

```
class Spooler {  
    Collection<?> task;  
  
    void setTask( Collection<?> c ) {  
        task = c;  
    }  
  
    void print( ) {  
        for ( Object e : task ) { System.out.println( e ); }  
    }  
}
```

Cannot be simulated  
with method type  
parameters

Correct: type checker  
instantiates task's type  
argument with c's

Works because  
every Java object  
has toString method

# Constrained Wildcards

```
static void printFormatted( Collection<?> c ) {  
    for ( Object e : c ) {  
        String s = e.format( 80 );  
        System.out.println( s );  
    }  
}
```

Type error: elements  
might not support  
method format

# Constrained Wildcards: Upper Bounds

```
interface Format {  
    String format( int width );  
}
```

```
static void printFormatted( Collection<? extends Format> c ) {  
    for ( Format e : c ) {  
        String s = e.format( 80 );  
        System.out.println( s );  
    }  
}
```

Typecheck under the  
assumption  
? <: Format

```
Collection<Object> c = new ArrayList<Object>( );  
printFormatted( c );
```

Compile-time error:  
Object is not a subtype of  
the upper bound Format

# More Bounded Wildcards

```
class Cell<T> {  
    T value;  
  
    void copyFromT( Cell<T> other ) {  
        value = other.value;  
    }  
  
    void copyFrom( Cell<? extends T> other ) {  
        value = other.value;  
    }  
  
    void copyTo( Cell<? super T> other ) {  
        other.value = value;  
    }  
}
```

Typecheck under  
the assumption

? <: T

Typecheck under  
the assumption

T <: ?

Wildcard can  
also have  
lower bounds

# Use-Site Variance: Covariance

- Wildcards allow clients to decide on variance at the use site
  - As opposed to Scala's declaration-site variance

```
class Random<T> {  
  T next( ) { ... }  
  void initialize( T i ) { ... }  
}
```

Java

Upper bound permits  
covariant instantiation

```
Random<? extends Person> r =  
    new Random<Student>( );  
Person a = r.next( );
```

Covariant uses  
typecheck

```
r.initialize( new Student( ) );
```

Contravariant uses do  
not typecheck



# Use-Site Variance: Contravariance

```
class OutputChannel<T> {  
  void write( T x ) { ... }  
  T lastWritten( ) { ... }  
}
```

Java

```
OutputChannel<? super Student> o =  
    new OutputChannel<Person>( );  
o.write( new Student( ) );
```

Lower bound permits  
contravariant  
instantiation

Contravariant uses  
typecheck

```
Person s = o.lastWritten( );
```

Covariant uses do not  
typecheck

# TreeSet Revisited

```
interface Comparator<T> {  
    int compare( T fst, T snd );  
}
```

```
class TreeSet<E> {  
    TreeSet( Comparator<? super E> c ) { ... }  
    ... }
```

TreeSet needs to compare  
**at least** set elements  
(use-site contravariance)

```
class Person { ... }
```

```
class Student extends Person { ... }
```

```
class PersonComp implements Comparator<Person> {  
    int compare( Person fst, Person snd ) { ... }  
}
```

```
TreeSet<Student> s = new TreeSet<Student>( new PersonComp() );
```

Wildcard instantiated with Person,  
which is a supertype of Student

# Wildcards vs. Method Type Parameters

```
void copyFrom( Cell<? extends T> other ) {  
    value = other.value;  
}
```

```
<S extends T> void copyFrom( Cell<S> other ) {  
    value = other.value;  
}
```

```
void copyTo( Cell<? super T> other ) {  
    other.value = value;  
}
```

```
<S super T> void copyTo( Cell<S> other ) {  
    other.value = value;  
}
```

```
Cell<String> s;  
Cell<Object> o;  
s = new Cell<String>();  
o = new Cell<Object>();  
o.copyFrom( s );
```

Identical client code:  
instantiations of  
wildcard and method  
type argument are  
inferred

Java does not support  
lower bounds for type  
parameters

# Wildcards vs. Class Type Parameters

```
class Wrapper {  
    Cell<?> data;  
}
```

Instantiation can  
change over time

```
Wrapper w = new Wrapper( );  
w.data = new Cell<String>( );  
w.data = new Cell<Object>( );
```

```
class Wrapper<T> {  
    Cell<T> data;  
}
```

```
Wrapper<Object> w = new Wrapper<Object>( );  
w.data = new Cell<String>( );  
w.data = new Cell<Object>( );
```

With type argument,  
instantiation is fixed  
when object is created

# Subtyping and Generics: Wildcards

- The bounds for a wildcard determine the set of possible instantiations

```
Cell<? extends Person> c;  
Cell<? super PhDStudent> d;
```

- For types S and T with the same class or interface, S is a subtype of T if for each type argument, the set of possible instantiations for S is a subset of the set of possible instantiations for T

```
c = new Cell<Student>( );
```

Instantiation is fixed  
(singleton set)

```
Cell<? extends Student> e = ...;  
c = e;
```

# Decidability

- Whether S is a subtype of T is undecidable in Java due to wildcards

```
class T { }  
  
class N<Z> { }  
  
class C<X> extends N<N<? super C<C<X>>>> {  
    public N<? super C<T>> foo( C<T> c ) {  
        return c;  
    }  
}
```

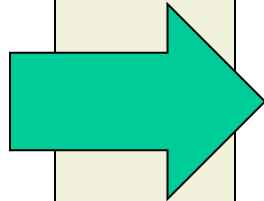
- In fact, Java generics are Turing-complete

# Type Erasure

- Java introduced generics in version 1.4
- For **backwards compatibility**, Sun did not want to change the virtual machine
- **Generic type information is erased** by compiler
  - $C<T>$  is translated to  $C$
  - $T$  is translated to its upper bound
  - Casts are added where necessary
- Only one classfile and only one class object to represent all instantiations of a generic class

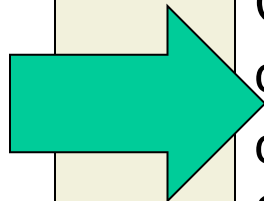
# Type Erasure: Example

```
class Cell<T extends Object> {  
    T value;  
  
    void set( T v ) {  
        value = v;  
    }  
  
    T get( ) {  
        return value;  
    }  
}
```



```
class Cell {  
    Object value;  
  
    void set( Object v ) {  
        value = v;  
    }  
  
    Object get( ) {  
        return value;  
    }  
}
```

```
Cell<String> c;  
c = new Cell<String>( );  
c.set( "Hello" );  
String s = c.get( );
```



```
Cell c;  
c = new Cell( );  
c.set( "Hello" );  
String s = ( String ) c.get( );
```



# Erasure: Missing Run-Time Information

```
void foo( Cell<?> c ) {  
    if( c instanceof Cell<String> )  
        ...  
}
```

Java

Compile-time error:  
generic types not  
allowed with **instanceof**

```
Class c = Cell<String>.class;
```

Java

Compile-time error:  
class object of generic  
types not available

```
Cell<String>[ ] a;  
a = new Cell<String>[10];
```

Java

Compile-time error:  
arrays of generic types  
not allowed

# Run-Time Information for Generics in C#

```
void Foo( object c ) {  
    if( c is Cell<string> )  
        ...  
}
```

C# does not  
have wildcards

C# can perform  
dynamic type test

C#

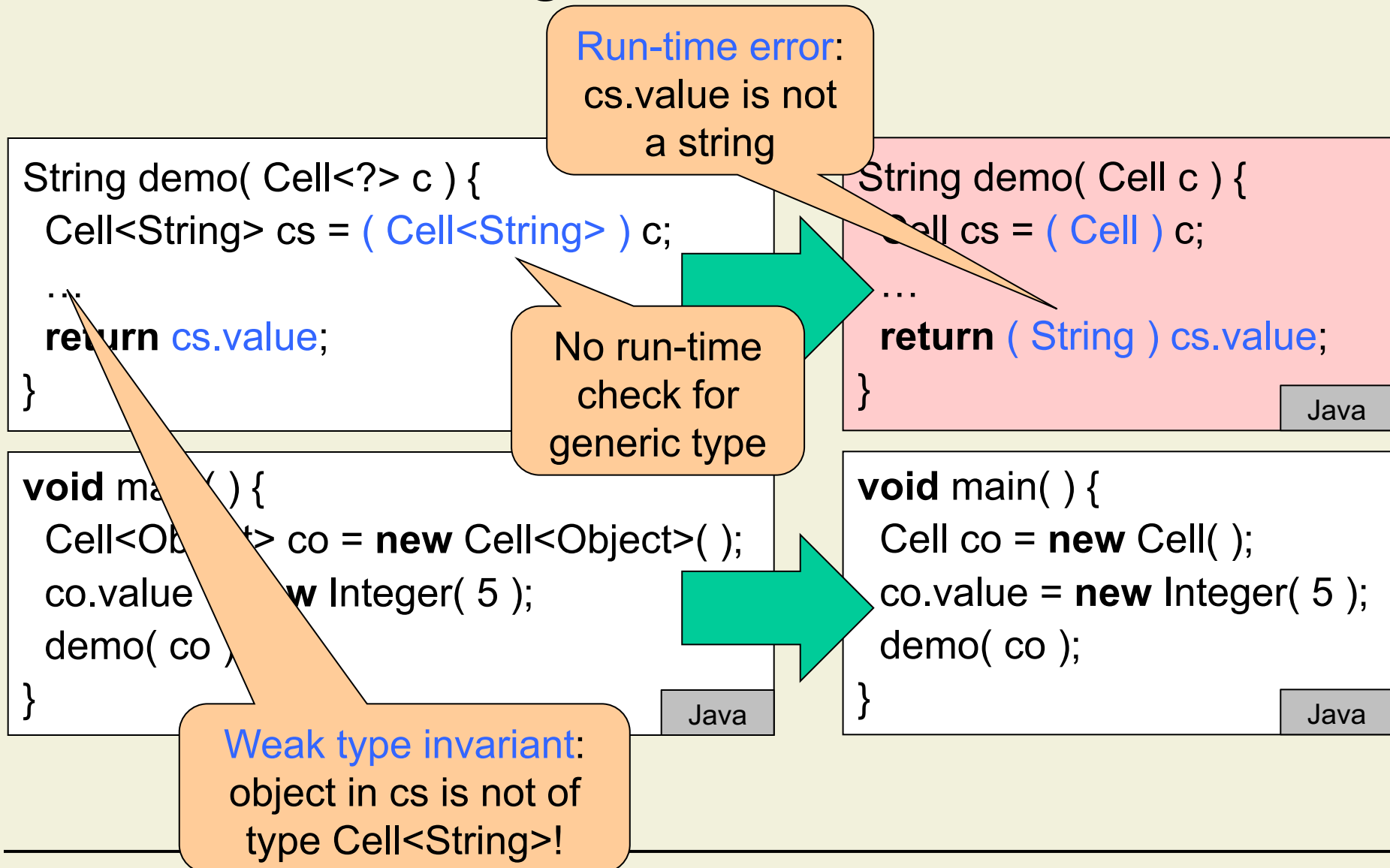
```
System.Type type = typeof( Cell<string> );
```

C# has run-time  
representation

```
Cell<String>[ ] a;  
a = new Cell<string>[10];
```

C# can perform run-time  
check for array update

# Erasure: Missing Run-Time Checks



# Erasure: Overloading

- Erasure may affect method signatures

```
class Erasure<S, T> {  
    void foo( S p ) {}  
    void foo( T p ) {}  
}
```

Java

Comparison

```
class Erasure<S, T> {  
    void foo( S p ) {}  
    void foo( T p ) {}  
}
```

C#

```
class Erasure<S, T extends Person> {  
    void foo( S p ) {}  
    void foo( T p ) {}  
}
```

Java

- Erasure in Java is a “leaky abstraction”

# Erasure: Static Fields

- In Java, static fields are shared by all instantiations of a generic class

```
class Count<T> {  
    static int c = 0;  
}
```

```
Count.c = 1;  
Count.c = 2;
```

Java

```
Count<string>.c = 1;  
Count<object>.c = 2;
```

C#

# C++ Templates

- Templates allow classes and methods to be **parameterized**
- Clients provide instantiations for template parameters

```
template<class T> class Queue {  
    T elem;  
    Queue<T>* next;  
public:  
    void enqueue( T e ) { ... };  
    T dequeue( ) { ... };  
};
```

C++

```
Queue<int> *q;  
q = new Queue<int>( );  
int s = 5;  
q->enqueue( s );  
int t = q->dequeue( );
```

C++

```
template<class T> void fill( T a[ ], T v )  
{ ... };
```

C++

# Template Instantiation

```
template<class T> class Queue {  
    T elem;  
    Queue<T>* next;  
    ...  
};
```

Compiler generates  
class for given  
template instantiation

```
class Queueint {  
    int elem;  
    Queueint* next;  
    ...  
};
```

Type checking is  
done for generated  
class, not for template

```
Queue<int> *q;  
...
```

Template Instantiation

```
Queueint *q;  
...
```

Client code uses  
generated class

# Templates and Type Checking

```
template<class T> class Queue {  
    T elem; Queue<T>* next;  
public:  
    void enqueue( T e ) {  
        if( next == NULL ) {  
            elem = e; next = new Queue<T>( );  
        } else {  
            if( e.compareTo( elem ) <= 0 )  
            { next->enqueue( elem ); elem = e; }  
            else next->enqueue( e );  
        }  
    };  
    T dequeue( ) { return "Hello"; };  
};
```

Compiler does not  
check availability  
of methods

Compiler does not type  
check template code

```
Queue<int> *q;  
q = new Queue<int>( );
```

Compiles even  
though template  
is instantiated

Compile-time errors:  
template methods  
not type correct

```
Queue<int> *q;  
q = new Queue<int>( );  
int s = 5;  
q->enqueue( s );  
int t = q->dequeue( );
```



# Templates and Type Checking (cont'd)

- Template code is type checked when instantiated
  - Type errors are not detected before instantiation
- No need for upper bounds on type parameters
  - Availability of methods is not checked anyway
  - Template has to document (informally) what it expects from its type arguments
- Different instantiations of templates are unrelated
  - Use template methods to write polymorphic methods
- Templates do not require run-time support
  - Run-time types correspond to generated classes

# Concepts

```
template<class T> concept Comparable =  
    requires( T a, T b ) {  
        { a.compareTo( b ) } -> std::same_as<bool>;  
    };
```

```
template<class T>  
    requires Comparable<T>  
    class Queue {  
        T elem; Queue<T>* next;  
        ...  
    };
```

```
Queue<int> *q;  
q = new Queue<int>( );
```

- Concepts declare syntactic and type constraints
- Can be used to express **structural upper bounds** on template args.
- Compiler checks that bounds are satisfied by instantiations

# Concepts (cont'd)

```
template<class T> class Queue {  
    T elem; Queue<T>* next;  
public:  
    void enqueue( T e ) {  
        if( next == NULL ) {  
            elem = e; next = new Queue<T>( );  
        } else {  
            if( e.compareTo( elem ) <= 0 )  
            { next->enqueue( elem ); elem = e; }  
            else next->enqueue( e );  
        }  
    };  
};
```

- Use of concepts is not enforced
- Templates without upper bounds still type check (backwards compatibility)
- Modularity issue persists

# Template Meta-Programming

```
template<int n> class Fact {  
public:  
    static const int val = Fact<n-1>::val * n;  
};
```

Template parameters  
need not be types

```
template<> class Fact<0>  
public:  
    static const int val = 1;  
};
```

Templates can  
be specialized

Compiler generates  
these instantiations

Through constant  
propagation, values  
are **computed by  
compiler**

```
int main( ) {  
    printf( "fact 3 = %d\n", Fact<3>::val );  
    printf( "fact 4 = %d\n", Fact<4>::val );  
    printf( "fact 5 = %d\n", Fact<5>::val );  
    return 0;  
}
```

# Generic Types vs. Templates: Summary

## Generic Types

- **Modular type checking** of generic class
  - Overhead (e.g., upper bounds)
- **Run-time support** desirable
- Typically no meta-programming

## Templates

- Type checking per instantiation
  - Flexibility like with structural typing
- No need for run-time support
- **Meta-programming** is Turing-complete

# References

- Xavier Leroy: *Java Bytecode Verification: Algorithms and Formalizations*. Journal of Automated Reasoning, 2003
- Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley: *The Java Virtual Machine Specification*. 2013  
<http://docs.oracle.com/javase/specs/>
- Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler: *Making the Future Safe for the Past: Adding Genericity to the Java Programming Language*. OOPSLA 1998
- Radu Grigore: *Java Generics are Turing Complete*. POPL 2017