

# Generic programming in OO Languages

## Reading

Text: Sections 9.4.1 and 9.4.3

J Koskinen, Metaprogramming in C++, Sections 2 – 5

Gilad Bracha, Generics in the Java Programming Language

# Questions

- If subtyping and inheritance are so great, why do we need type parameterization in object-oriented languages?
  - The great polymorphism debate
    - Subtype polymorphism
      - Apply  $f(\text{Object } x)$  to any  $y : C <: \text{Object}$
    - Parametric polymorphism
      - Apply  $\text{generic } <T> f(T x)$  to any  $y : C$
- Do these serve similar or different purposes?

# Outline

- C++ Templates



- Polymorphism vs Overloading

- C++ Template specialization

- Example: Standard Template Library (STL)

- C++ Template metaprogramming

- Java Generics

- Subtyping versus generics

- Static type checking for generics

- Implementation of Java generics

# Polymorphism vs Overloading

- Parametric polymorphism
  - Single algorithm may be given many types
  - Type variable may be replaced by *any* type
  - $f :: t \rightarrow t \Rightarrow f :: \text{Int} \rightarrow \text{Int}, f :: \text{Bool} \rightarrow \text{Bool}, \dots$
- Overloading
  - A single symbol may refer to more than one algorithm
  - Each algorithm may have different type
  - Choice of algorithm determined by type context
  - Types of symbol may be arbitrarily different
  - $+$  has types  $\text{int} * \text{int} \rightarrow \text{int}, \text{real} * \text{real} \rightarrow \text{real}, \dots$

# Polymorphism: Haskell vs C++

- Haskell polymorphic function
  - Declarations (generally) require no type information
  - Type inference uses type variables
  - Type inference substitutes for variables as needed to instantiate polymorphic code
- C++ function template
  - Programmer declares argument, result types of fctns
  - Programmers use template, explicit type parameters
  - Function application: type checker does instantiation

# Example: swap two values

- Haskell

```
swap :: (IORef a, IORef a) -> IO ()
swap (x,y) = do {
    val_x <- readIORef x; val_y <- readIORef y;
    writeIORef y val_x; writeIORef x val_y;
    return () }
```

- C++

```
template <typename T>
void swap(T& x, T& y){
    T tmp = x; x=y; y=tmp;
}
```

Haskell, C++ polymorphic functions both swap two values of any type, but they are compiled very differently


# Implementation

- Haskell
  - Swap is compiled into one function
  - Typechecker determines how function can be used
- C++
  - Swap is instantiated at a form of compile time
  - Separate copy of compiled code for each type of use
- Why the difference?
  - Haskell reference cell is passed by pointer, local variables are pointers to values on the heap
  - C++ arguments passed by reference (pointer), but local x is on stack and its size depends on its type

# Implicit constraints on type parameter

- Example: polymorphic sort function


```
template <typename T>
void sort( int count, T * A[count] ) {
    for (int i=0; i<count-1; i++)
        for (int j=i+1; j<count-1; j++)
            if (A[j] < A[i]) swap(A[i],A[j]);
}
```



- How does instantiation depend on type T?
  - Indexing into array
  - Meaning and implementation of <



# Outline

- C++ Templates
  - Polymorphism vs Overloading
  -  C++ Template specialization
    - Example: Standard Template Library (STL)
    - C++ Template metaprogramming
- Java Generics
  - Subtyping versus generics
  - Static type checking for generics
  - Implementation of Java generics

# Partial specialization

- Example: general swap can be inefficient

```
template <class T>  
    void swap ( T& a, T& b ) { T c=a; a=b; b=c; }
```

- Specialize general template

```
template <class T>  
    void swap(vector<T>&, vector<T>&);  
// implement by moving pointers in vector headers in const time
```

- Advantage
  - Use better implementation for specific kinds of types
  - Intuition: “overloaded” template
  - Compiler chooses most specific applicable template

# Another example

```
/* Primary template */
    template <typename T> class Set {
        // Use a binary tree
    };


/* Full specialization */
    template <> class Set<char> {
        // Use a bit vector
    };

/* Partial specialization */
    template <typename T> class Set<T*> {
        // Use a hash table
    };
```

# C++ Template implementation

- Compile-time instantiation
  - Compiler chooses template that is best match
    - There can be more than one applicable template
  - Template instance is created
    - Similar to syntactic substitution of parameters ( $\beta$ -reduction)
    - Can be done after parsing, etc. (we will ignore details)
  - Overloading resolution *after* substitution
- Limited forms of “separate compilation”
  - Overloading, data size restrict separate compilation
  - Several models – details tricky, not needed for CS242

# Outline

- C++ Templates
  - Polymorphism vs Overloading
  - C++ Template specialization
  -  Example: Standard Template Library (STL)
    - C++ Template metaprogramming
- Java Generics
  - Subtyping versus generics
  - Static type checking for generics
  - Implementation of Java generics

# Standard Template Library for C++

- Many generic abstractions
  - Polymorphic abstract types and operations
- Useful for many purposes
  - Excellent example of generic programming
- Efficient running time (not always space efficient)
- Written in C++
  - Uses template mechanism and overloading
  - Does not rely on objects ★

Architect: Alex Stepanov,  
previous work with D Musser ...

# Main entities in STL

- Container: Collection of typed objects
  - Examples: array, list, associative dictionary, ...
- Iterator: Generalization of pointer or address
- Algorithm
- Adapter: Convert from one form to another
  - Example: produce iterator from updatable container
- Function object: Form of closure
- Allocator: encapsulation of a memory pool
  - Example: GC memory, ref count memory, ...

# Example of STL approach

- Function to merge two sorted lists

- $\text{merge} : \text{range}(s) \times \text{range}(t) \times \text{comparison}(u)$   
→  $\text{range}(u)$

This is conceptually right, but not STL syntax

- Basic concepts used

- $\text{range}(s)$  - ordered “list” of elements of type  $s$ , given by pointers to first and last elements
  - $\text{comparison}(u)$  - boolean-valued function on type  $u$
  - $\text{subtyping}$  -  $s$  and  $t$  must be subtypes of  $u$





# Comparing STL with other libraries

- C:

```
qsort( (void*)v, N, sizeof(v[0]), compare_int );
```

- C++, using raw C arrays:

```
int v[N];
```

```
sort( v, v+N );
```

- C++, using a vector class:

```
vector v(N);
```

```
sort( v.begin(), v.end() );
```

# Efficiency of STL


- Running time for sort

	N = 50000	N = 500000
C	1.4215	18.166
C++ (raw arrays)	0.2895	3.844
C++ (vector class)	0.2735	3.802

- Main point

- Generic abstractions can be convenient and efficient !
- But watch out for code size if using C++ templates...

# Outline

- C++ Templates
  - Polymorphism vs Overloading
  - C++ Template specialization
  - Example: Standard Template Library (STL)
  -  C++ Template metaprogramming
- Java Generics
  - Subtyping versus generics
  - Static type checking for generics
  - Implementation of Java generics

# C++ Template Metaprogramming

- Explicit parametric polymorphism
- Maximal typing flexibility
  - Allow template use whenever instance of template will compile
- Specialization and partial specialization
  - Compiler chooses best match among available templates
- Allow different kinds of parameters
  - Type parameters, template parameters, and non-type parameters (integers, ...)
- Support mixins
  - Class templates may inherit from a type parameter
- Support template meta-programming techniques
  - Conditional compilation, traits, ...
  - See books, courses, web sites (Keith Schwarz' CS106L)

# Metaprogramming example

```
template<int N> struct Factorial {  
    enum{ value = Factorial<N-1>::value * N};  
};  
template<> struct Factorial<0>{  
    enum{value = 1};  
};  
int main(){  
    char array[Factorial<4>::value];  
    std::cout << sizeof(array);  
}
```

Reference: J Koskinen, Metaprogramming in C++,  
<http://www.cs.tut.fi/~kk/webstuff/MetaprogrammingCpp.pdf>

# Policy Class Example

- Policy class
  - A policy class implements a particular behavior
- Sample code that we will parameterize by policies

```
template <typename T>
class Vector{
    public:
        /* ... ctors, dtor, etc. */
        T& operator[] (size_t);
        const T& operator[] (size_t) const;
        void insert(iterator where, const T& what);
        /* ... etc. ... */
};
```

# Vector template with policies

```
template <typename T,  
          typename RangePolicy,  
          typename LockingPolicy>  
class Vector : public RangePolicy,  
              public LockingPolicy  
...  
T& Vector<T, RangePolicy, LockingPolicy>::  
operator[] (size_t position){  
    LockingPolicy::Lock lock; ← Lock using locking policy  
    RangePolicy::CheckRange(position, this->size);  
    return this->elems[position]; ↑ Check range using range policy  
}
```

Class parameters used as base classes are sometimes called “mixins”



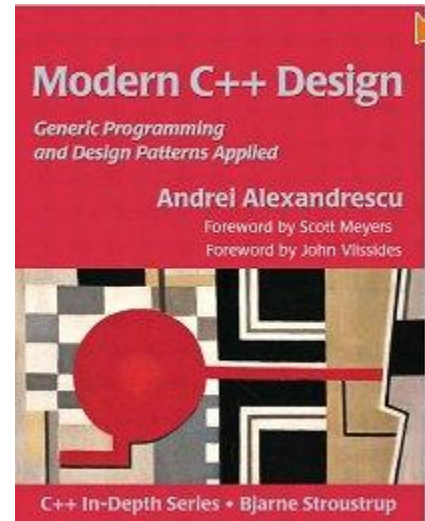
# Sample range policy

```
class ThrowingErrorPolicy{
protected:
    ~ThrowingErrorPolicy() {}
    static void CheckRange(size_t pos, size_t numElems){
        if(pos >= numElems)
            throw std::out_of_bounds("Bad!");
    }
};
```

Alternate: log error without raising an exception

# Many other metaprogramming ideas

- Policy-based class design
- Type lists and type selection
- Combining metaprogramming and design patterns



# Outline

- C++ Templates
  - Polymorphism vs Overloading
  - C++ Template specialization
  - Example: Standard Template Library (STL)
  - C++ Template metaprogramming
- Java Generics
  - ➔ Subtyping versus generics
    - Static type checking for generics
    - Implementation of Java generics

# 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 generics
  - No parametric polymorphism
  - Many considered this the biggest deficiency of Java
- Java type system does not let you “cheat”
  - Can cast “down” 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++, 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();
```

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

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

# Why no generics in early Java ?

- Many proposals
- Basic language goals seem clear
- Details take some effort to work out
  - Exact typing constraints
  - Implementation
    - Existing virtual machine?
    - Additional bytecodes?
    - Duplicate code for each instance?
    - Use same code (with casts) for all instances

Java Community proposal (JSR 14) incorporated into Java 1.5

# JSR 14 Java Generics (Java 1.5, “Tiger”)

- Adopts syntax on previous slide
- Adds auto boxing/unboxing

User conversion


```
Stack<Integer> st =  
    new Stack<Integer>();  
st.push(new Integer(12));  
...  
int i = (st.pop()).intValue();
```

Automatic conversion

```
Stack<Integer> st =  
    new Stack<Integer>();  
st.push(12);  
...  
int i = st.pop();
```



# Outline

- C++ Templates
  - Polymorphism vs Overloading
  - C++ Template specialization
  - Example: Standard Template Library (STL)
  - C++ Template metaprogramming
- Java Generics
  - Subtyping versus generics
  -  Static type checking for generics
  - Implementation of Java generics

# Java generics are type checked

- A generic class may use operations on objects of a parameter type
  - Example: `PriorityQueue<T> ... if x.less(y) then ...`
- Two possible solutions
  - C++: Compile and see if operations can be resolved
  - Java: Type check and compile generics independently
    - May need additional information about type parameter
      - What methods are defined on parameter type?
      - Example: `PriorityQueue<T extends ...>`

# Example

- Generic interface

```
interface Collection<A> {  
    public void add (A x);  
    public Iterator<A> iterator ();  
}
```

```
interface Iterator<E> {  
    E next();  
    boolean hasNext();  
}
```

- Generic class implementing Collection interface

```
class LinkedList<A> implements Collection<A> {  
    protected class Node {  
        A elt;  
        Node next = null;  
        Node (A elt) { this.elt = elt; }  
    }  
    ...  
}
```

# Wildcards

- Example

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

- Meaning: Any representative from a family of types

- unbounded wildcard **?**

- matches all types

- lower-bound wildcard **? extends Supertype**

- matches all types that are subtypes of Supertype

- upper-bound wildcard **? super Subtype**

- matches all types that are supertypes of Subtype

# Type concepts for understanding Generics

- Parametric polymorphism

–  $\text{max} : \forall t \ ((t \times t) \rightarrow \text{bool}) \rightarrow ((t \times t) \rightarrow t)$

given lessThan function

return max of two arguments

- Bounded polymorphism

–  $\text{printString} : \forall t \ <: \text{Printable} . t \rightarrow \text{String}$

for every subtype t of Printable

function from t to String

- F-Bounded polymorphism

–  $\text{max} : \forall t \ <: \text{Comparable}(t) . t \times t \rightarrow t$

for every subtype t of ...

return max of object and argument

# F-bounded subtyping

- Generic interface

```
interface Comparable<T>{ public int compareTo(T arg);}
    x.compareTo(y) = negative, 0, positive if y is <=> x
```

- Subtyping

```
interface A { public int compareTo(A arg);
              int anotherMethod (A arg); ... }
```

<:

```
interface Comparable<A>
    = { public int compareTo(A arg);}
```



# Example static max method

- Generic interface

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

- Example

```
public static <T extends Comparable<T>> T max(Collection<T> coll) {  
    T candidate = coll.iterator().next();  
    for (T elt : coll) {  
        if (candidate.compareTo(elt) < 0) candidate = elt;  
    }  
    return candidate;  
}
```

candidate.compareTo :  $T \rightarrow \text{int}$

# This would typecheck without F-bound ...

- Generic interface

```
interface Comparable<T> { public int compareTo(T arg); ... }
```

Object



- Example

```
public static <T extends Comparable<T>> T max(Collection<T> coll) {  
    T candidate = coll.iterator().next();  
    for (T elt : coll) {  
        if (candidate.compareTo(elt) < 0) candidate = elt;  
    }  
    return candidate;  
}
```

Object



candidate.compareTo : T → int

How could you write an implementation of this interface?



# Generics are *not* co/contra-variant

- Array example (review)

```
Integer[] ints = new Integer[] {1,2,3};
```

```
Number[] nums = ints;
```

```
nums[2] = 3.14; // array store -> exception at run time
```

- List example

```
List<Integer> ints = Arrays.asList(1,2,3);
```

```
List<Number> nums = ints; // compile-time error
```

– Second does not compile because

```
List<Integer> <: List<Number>
```

# Return to wildcards

- Recall example

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

- Compare to

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

- This version is *much* less useful than the one above
  - Wildcard allows call with kind of collection as a parameter,
  - Alternative only applies to `Collection<Object>`, not a supertype of other kinds of collections!

# Outline

- C++ Templates
  - Polymorphism vs Overloading
  - C++ Template specialization
  - Example: Standard Template Library (STL)
  - C++ Template metaprogramming
- Java Generics
  - Subtyping versus generics
  - Static type checking for generics
  - ➔ Implementation of Java generics

# Implementing Generics

- Type erasure
  - Compile-time type checking uses generics
  - Compiler eliminates generics by erasing them
    - Compile `List<T>` to `List`, `T` to `Object`, insert casts
- “Generics are not templates”
  - Generic declarations are typechecked
  - Generics are compiled once and for all
    - No instantiation
    - No code expansions

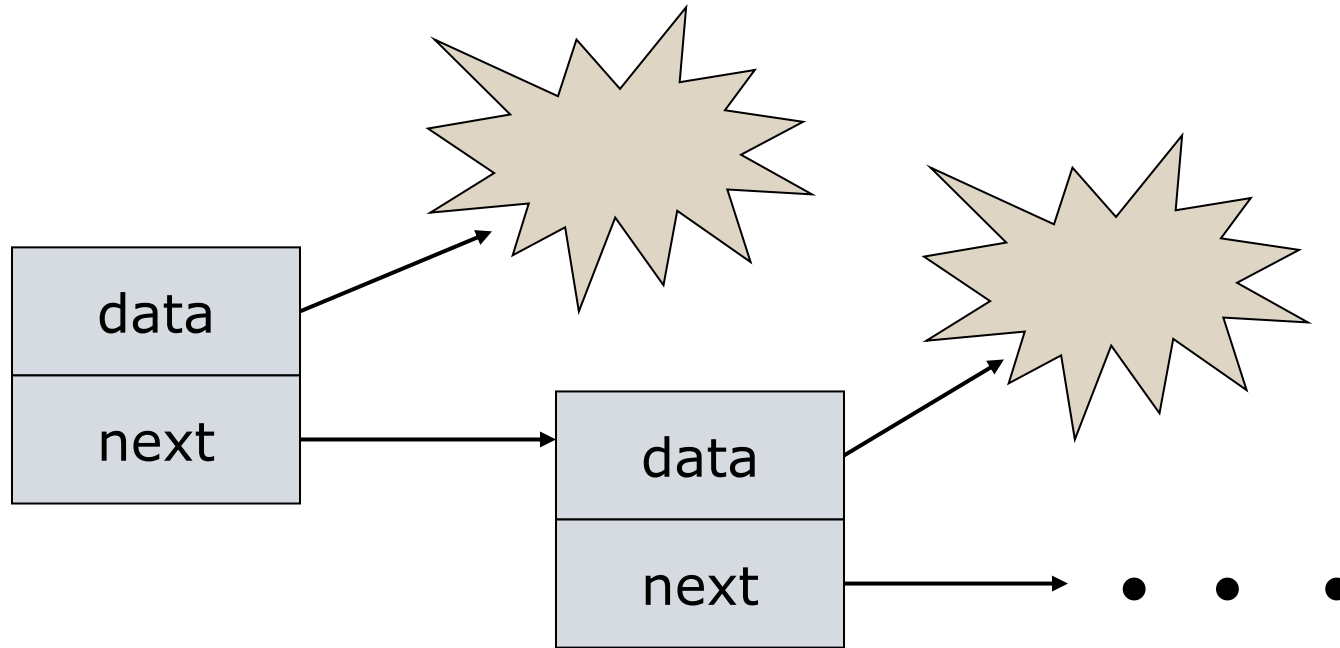
# Implementation Options

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

- Example for next few slides: generic list class

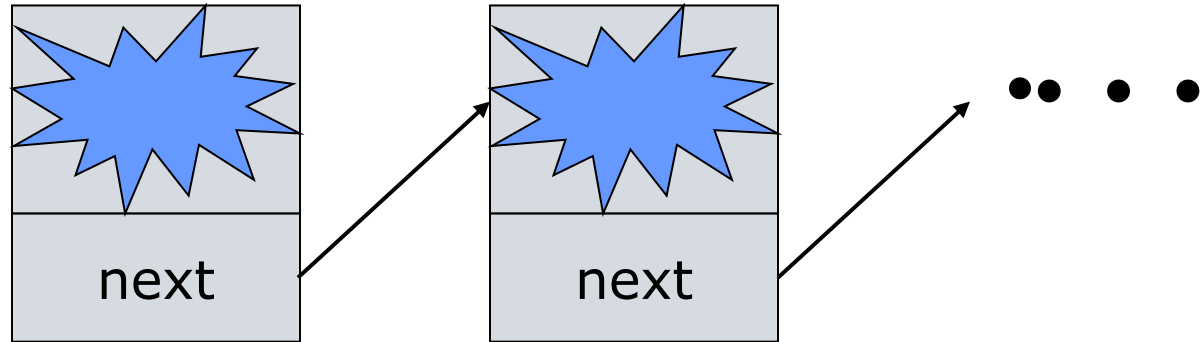
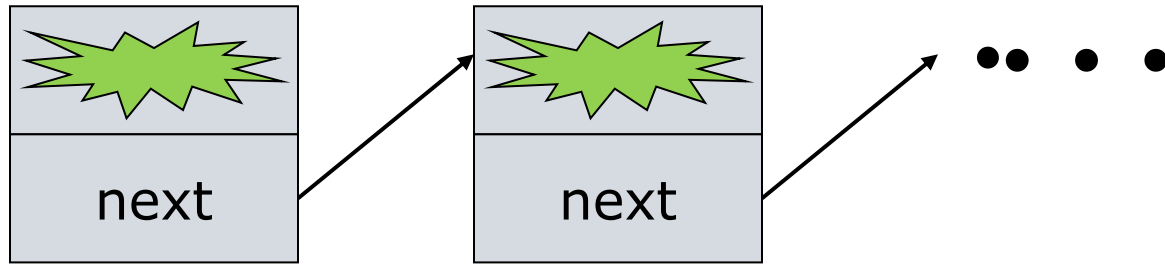
```
template <type t> class List {  
    private: t* data; List<t> * next;  
    public: void    Cons (t* x) { ... }  
            t*     Head (    ) { ... }  
            List<t> Tail (    ) { ... }  
};
```

# “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 the same size
  - We can use the same representation, code for all types
- Data stored in local variables (C++)
  - Each list cell must have space for data
  - Different representation needed for different types
  - Different code if offset of fields is built into code
- When is template instantiated?
  - Compile- or link-time (C++)
  - Java alternative: class-load-time generics (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 `G<param>`
  - Check use of parameter type according to constraints
  - Produce extended form of bytecode class file
    - Store constraints, type parameter names in bytecode file
- Instantiate when class `G<actual>` is loaded
  - Replace parameter type by actual class
  - Result can be transformed to 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    GetHashCode ();  
};
```

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

```
Method void Insert(Name, Integer)  
  aload_1  
  invokevirtual #6 <Method Name.GetHashCode()I>  
  istore_3  aload_0  iload_3  aload_1  aload_2  
  invokevirtual #7 <Method HashTable<Name,Integer>  
    InsertAt(ILName;LInteger;)V>  
  return
```

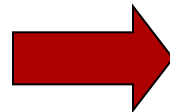
# Loading parameterized class file

- Use of `HashTable <Name, Integer>` starts 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 “Erasure” 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` !
    - Create new object of parameter type 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



# Outline

- C++ Templates
    - Polymorphism vs Overloading
    - C++ Template specialization
    - Example: Standard Template Library (STL)
    - C++ Template metaprogramming
  - Java Generics
    - Subtyping versus generics
    - Static type checking for generics
    - Implementation of Java generics
- Comparison (next slide )

# Comparison

	Templates	Generics
Type parameterization	Classes and functions may have type parameters.	Classes and methods may have type parameters.
Flexibility	Compile-time instantiation allows checking and overload resolution at compile time.	Separate compilation using type constraints supplied by the programmer.
Specialization	Both template specialization and partial specialization. Compiler chooses the best match.	No specialization or partial specialization.
Non-type parameters	Compile-time instantiation with integer parameters; optimize code at compile time.	No compile-time parameters.
Mixins	Class templates may use a type parameter as a base class.	Cannot inherit from type parameters

# Additional links for material not in book

- **Template metaprogramming**
  - <http://www.cs.tut.fi/~kk/webstuff/MetaprogrammingCpp.pdf>
- **Enhancements in JDK 5**
  - <http://docs.oracle.com/javase/1.5.0/docs/guide/language/index.html>
- **J2SE 5.0 in a Nutshell**
  - <http://www.oracle.com/technetwork/articles/javase/j2se15-141062.html>
- **Generics**
  - <http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.pdf>

