CS 242

The Java Programming Language

John Mitchell

Reading: Chapter 13 + Gilad Bracha, Generics in the Java Programming Language, Sun Microsystems, 2004 (see web site).

Outline

- Language Overview
  - History and design goals
- Classes and Inheritance
  - Object features
  - Encapsulation
  - Inheritance
- Types and Subtyping
  - Primitive and ref types
  - Interfaces; arrays
  - Exception hierarchy
- Generics
  - Subtype polymorphism. generic programming

-- next lecture (separate slides) --

- Virtual machine overview
  - Loader and initialization
  - Linker and verifier
  - Bytecode interpreter
- Method lookup
  - four different bytecodes
- Verifier analysis
- Implementation of generics
- Security
  - Buffer overflow
  - Java "sandbox"
  - Type safety and attacks

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 on heap, accessed through pointers
  - No functions, no multiple inheritance, no go to, no operator overloading, few automatic coercions
- Portability and network transfer
  - Bytecode interpreter on many platforms
- Reliability and Safety
  - Typed source and typed bytecode language
  - Run-time type and bounds checks
  - Garbage collection

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
  - Interoperability
    - Provision for "native" methods

Java Release History

- ◆1995 (1.0) – First public release
- ◆1997 (1.1) – Inner classes
- ◆2001 (1.4) – Assertions
  - Verify programmers understanding of code
- ◆2004 (1.5) – Tiger
  - Generics, foreach, Autoboxing/Unboxing,
  - Typesafe Enums, Varargs, Static Import,
  - Annotations, concurrency utility library

http://java.sun.com/developer/technicalArticles/releases/j2se15/

Improvements through Java Community Process

Enhancements in JDK 5 (= Java 1.5)

- ◆Generics
  - Polymorphism and compile-time type safety (JSR 14)
- ◆Enhanced for Loop
  - For iterating over collections and arrays (JSR 201)
- ◆Autoboxing/Unboxing
  - Automatic conversion between primitive, wrapper types (JSR 201)
- ◆Typesafe Enums
  - Enumerated types with arbitrary methods and fields (JSR 201)
- ◆Varargs
  - Puts argument lists into an array; variable-length argument lists
- ◆Static Import
  - Avoid qualifying static members with class names (JSR 201)
- ◆Annotations (Metadata)
  - Enables tools to generate code from annotations (JSR 175)
- ◆Concurrency utility library, led by Doug Lea (JSR-166)

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

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
  - Static typing => more efficient than Smalltalk
  - Dynamic linking, interfaces => slower than C++

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

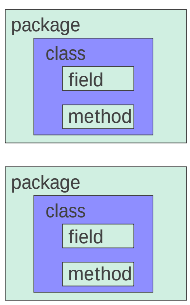
    We'll discuss later ...
- Cannot do this (would be bad C++ style anyway):
  - Obj* obj = (Obj*)malloc(sizeof(Obj));
- Static fields of class initialized at class load time
  - Talk about class loading later

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
    - Space overflow is not really the right condition to trigger finalization when an object holds a lock...)
  - Important convention: call super.finalize

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



Visibility and access

- Four visibility distinctions
  - public, private, protected, package
- 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)
  - java.lang.String.substring()

    package class method

Inheritance

- Similar to Smalltalk, C++
- Subclass inherits from superclass
  - Single inheritance only (but Java has interfaces)
- Some additional features
  - Conventions regarding *super* in constructor and *finalize* methods
  - Final classes and methods

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
 };
```

Class *Object*

- Every class extends another class
  - Superclass is *Object* if no other class named
- Methods of class *Object*
  - GetClass – return the Class object representing class of the object
  - ToString – returns string representation of object
  - equals – default object equality (not ptr equality)
  - hashCode
  - Clone – makes a duplicate of an object
  - wait, notify, notifyAll – used with concurrency
  - finalize

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

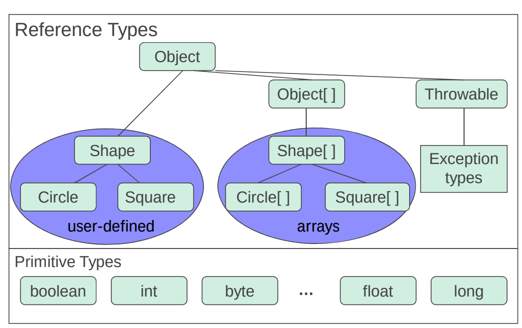
Outline

- 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

Classification of Java types



Subtyping

- Primitive types
  - Conversions: int -> long, double -> long, ...
- 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)
- Arrays
  - Covariant subtyping – not consistent with semantic principles

Java class subtyping

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

  How much conformance is needed in principle?
- Java rule
  - Java 1.1: Arguments and returns must have identical types, may remove exceptions
  - Java 1.5: covariant return type specialization

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
}
```

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

Array subtyping

- Covariance
  - if S <: T then S[ ] <: T[ ]
- Standard type error

```
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
```

Covariance problem again …

- Remember Simula problem
  - If A <: B, then A ref <: B ref
  - Needed run-time test to prevent bad assignment
  - Covariance for assignable cells is not right in principle
- Explanation
  - interface of “T reference cell” is

    put : $T \to T$ ref

    get : T ref $\to T$
  - Remember covariance/contravariance of functions

Afterthought on Java arrays

Date: Fri, 09 Oct 1998 09:41:05 -0600
From: bill joy
Subject: ...[discussion about java genericity]

actually, java array covariance was done for less noble reasons ...: it made some generic "bcopy" (memory copy) and like operations much easier to write...

I proposed to take this out in 95, but it was too late (...).

i think it is unfortunate that it wasn't taken out...

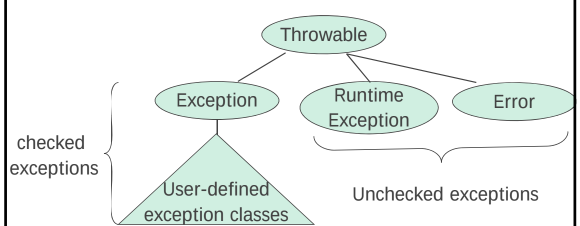
it would have made adding genericity later much cleaner, and [array covariance] doesn't pay for its complexity today.

wnj

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
  - Subtyping between exception classes
    - Use subtyping to match type of exception or pass it on …
    - Similar functionality to ML pattern matching in handler
  - 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

Try/finally blocks

- Exceptions are caught in try blocks

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

- Implementation: finally compiled to jsr

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 generics
  - 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

Java 1.0	Generics
class Stack {	class Stack<A> {
void push(Object o) { ... }	void push(A a) { ... }
Object pop() { ... }	A pop() { ... }
...}	...}
String s = "Hello";	String s = "Hello";
Stack st = new Stack();	Stack<String> st = new Stack<String>();
...	
st.push(s);	st.push(s);
...	...
s = (String) st.pop();	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	Automatic conversion
Stack<Integer> st = new Stack<Integer>();	Stack<Integer> st = new Stack<Integer>();
st.push(new Integer(12));	st.push(12);
...	...
int i = (st.pop()).intValue();	int i = st.pop();

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++: Link and see if all operations can be resolved
  - Java: Type check and compile generics w/o linking
    - 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   ?
    - all types
  - lower-bound wildcard   ? extends Supertype
    - all types that are subtypes of Supertype
  - upper-bound wildcard   ? super Subtype
    - all types that are supertypes of Subtype

Type concepts for understanding Generics

- Parametric polymorphism
  - max : $\forall t\ \underbrace{((t \times t) \to bool)}_{\text{given lessThan function}} \to \underbrace{((t \times t) \to t)}_{\text{return max of two arguments}}$
- Bounded polymorphism
  - printString : $\underbrace{\forall t\ <:\ Printable}_{\text{for every subtype t of Printable}}\ .\ \underbrace{t \to String}_{\text{function from t to String}}$
- F-Bounded polymorphism
  - max : $\underbrace{\forall t\ <:\ Comparable\ (t)}_{\text{for every subtype t of ...}}\ .\ \underbrace{t \times t \to t}_{\text{return max of object and argument}}$

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 : $T \to int$

This would typecheck without F-bound ...

- Generic interface

Object

```
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;
}
```

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 old one
    - Wildcard allows call with kind of collection as a parameter,
    - Alternative only applies to Collection<Object>, not a supertype of other kinds of collections!

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 bloat"

More later when we talk about virtual machine ...

Additional links for material not in book

- Enhancements in JDK 5
  - http://java.sun.com/j2se/1.5.0/docs/guide/language/index.html
- J2SE 5.0 in a Nutshell
  - http://java.sun.com/developer/technicalArticles/releases/j2se15/
- Generics
  - http://www.langer.camelot.de/Resources/Links/JavaGenerics.htm