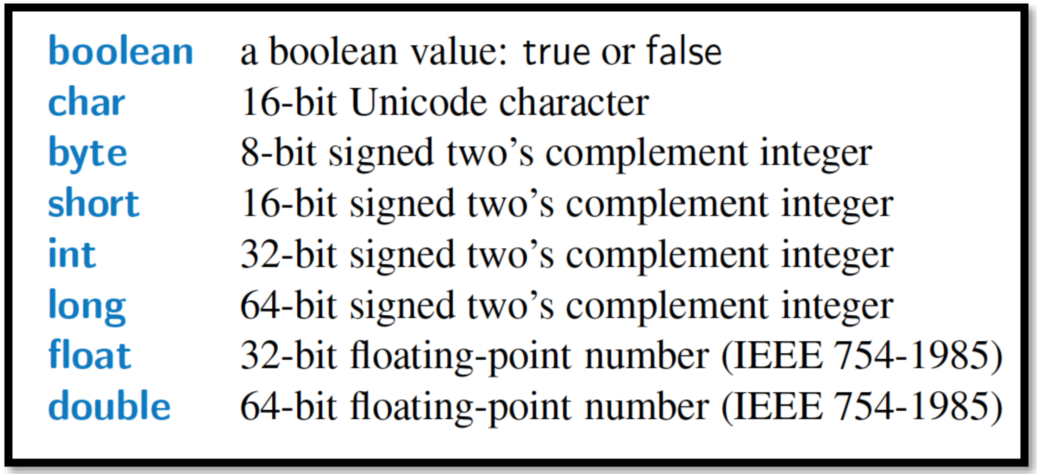
Introduction to Java

Sridhar Alagar
sridhar@utdallas.edu

Basic Types



boolean	a boolean value: true or false
char	16-bit Unicode character
byte	8-bit signed two's complement integer
short	16-bit signed two's complement integer
int	32-bit signed two's complement integer
long	64-bit signed two's complement integer
float	32-bit floating-point number (IEEE 754-1985)
double	64-bit floating-point number (IEEE 754-1985)

Classes

Each (outer) class must be in its own file, with name of file same as the class name (case sensitive). May contain nested classes.

Naming convention: starts with upper case letter; [A-Z][a-zA-Z]*

Constructor has same name as class, no return type; can be overloaded;
if not given, automatic constructor is supplied;
constructor cannot be called directly; called using “new”.

Fields declared to be “static” have only one instance (across all instances of the class).

Methods are “non-static” by default, and they work on a given class object.
<u>A static method cannot call a non-static method</u>

Example: Item class

```
public class Item {
    private int element;

    Item() { element = 0;}
    Item(int x) { element = x; }

    public int getItem() { return element; }

    public void setItem(int x) { element = x; }

    public int compareTo(Item another) {
        if (this.element < another.element) { return -1; }
        else if (this.element > another.element) { return 1; }
        else return 0; }

    public String toString(){return Integer.toString(element);}
}
```

All objects are references

All objects must be allocated with “`new`”:

```
Item i; i = new Item()
Item[] e = new Item[n];
```

It allocates an array `e` of `n` references to objects of type “`Item`”: `e[0..n-1]`. Space for the actual objects is not allocated.

Later, you could run

```
        e[i] = new Item(...);” or “e[i] = obj;
```

(where `obj` is some object of type `Item`, that has already been created), so that `e[i]` references an actual object.

Arrays are also references. Therefore,

```
        Item[] A, B; ....  A = B;
```

is legal, and `A` now references the same array as `B`.

main()

Execution begins with the `main()` function of the class with which execution is started.

So, if we run "`java Example`" it starts execution of `main()` in the file `Example.java`.

`main()` must be "`public static void`" and takes one array of strings as its parameter. Usual declaration:

```
      public static void main(String[] args) {... }
```

`args.length` is the number of command line arguments. The arguments are: `args[0]`,`args[1]`,…,`args[args.length-1]`.

I/O

Until you build some expertise, use “Scanner” class for reading input and send output to the console (stdout) using System.out:

```
import java.util.Scanner;
import java.io.FileNotFoundException;
import java.io.File;

public class IO {
    // Use file name from command line if given, else read from console.
    public static void main(String[] args) throws FileNotFoundException
    {  Scanner in;
        if (args.length > 0) {
            File inputFile = new File(args[0]);
            in = new Scanner(inputFile);
        } else { in = new Scanner(System.in); }

        int s = in.nextInt();   float t = in.nextFloat();
        System.out.println("s: " + s + " t: " + t);
    }
}
```

Wrapper Types

Many algorithms in Java library are designed to work with object types

To overcome this problem, Java has a wrapper class for all primitive types

Implicit conversion between a primitive type and its Wrapper type happens through automatic boxing and unboxing.

Example Wrapper Types

Base Type	*Class Name*	*Creation Example*	*Access Example*
boolean	Boolean	obj = new Boolean(true);	obj.booleanValue()
char	Character	obj = new Character('Z');	obj.charValue()
byte	Byte	obj = new Byte((byte) 34);	obj.byteValue()
short	Short	obj = new Short((short) 100);	obj.shortValue()
int	Integer	obj = new Integer(1045);	obj.intValue()
long	Long	obj = new Long(10849L);	obj.longValue()
float	Float	obj = new Float(3.934F);	obj.floatValue()
double	Double	obj = new Double(3.934);	obj.doubleValue()

```
int j = 8;
Integer a = new Integer(12);
int k = a;                          // implicit call to a.intValue()
int m = j + a;                      // a is automatically unboxed before the addition
a = 3 * m;                          // result is automatically boxed before assignment
Integer b = new Integer("-135");    // constructor accepts a String
int n = Integer.parseInt("2013");   // using static method of Integer class
```

Code hierarchy

Java classes are organized into packages, hierarchically.

Commonly used: `java.io`, `java.util`, `java.lang`

Code is organized into packages (name of folder in which the package files reside must be the name of the package).

Classes within a package get some additional access (like “friend” in C++).

For this course, create a folder whose name is your netid (e.g., axb012345), and use package declaration:

```
    package axb012345;
```

`CLASSPATH` environment variable is used to find other classes referenced in a program; “.” refers to the current directory.

Interfaces

Interfaces are syntactically like classes but contains only constants and abstract methods (without implementation).

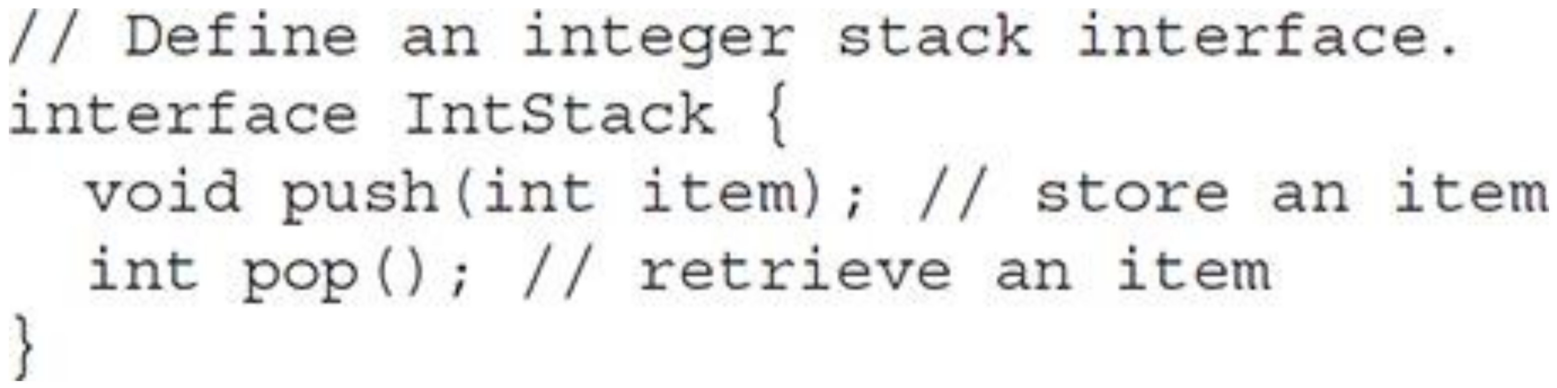
Abstract methods are like function prototypes in C/C++

Define interface but don’t make assumptions about how to implement it

A class can implement any number of interfaces

Some common interfaces: `List`, `Queue`, `Iterator`, `Comparable`

Interfaces - Example



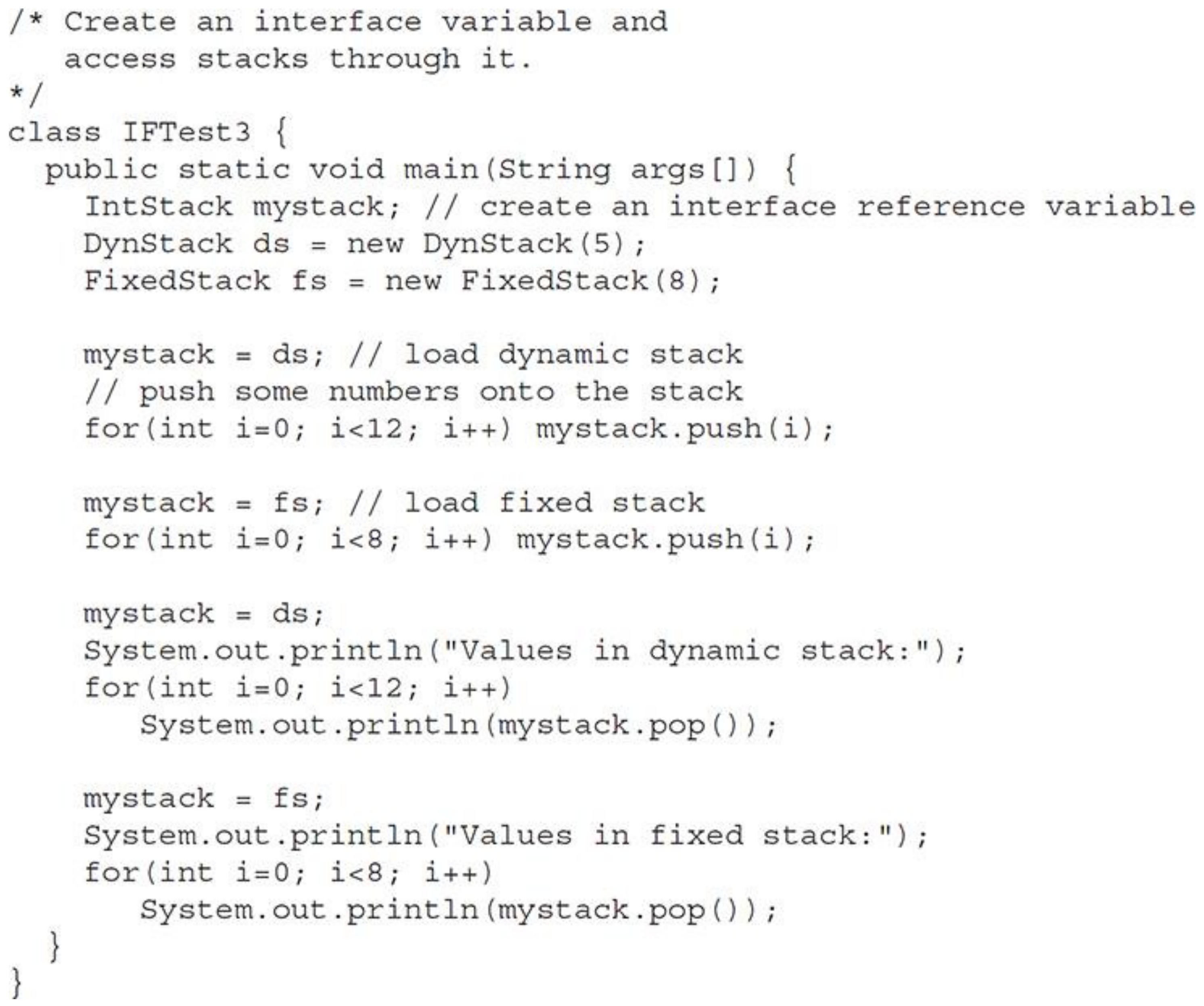
```
// Define an integer stack interface.
interface IntStack {
  void push(int item); // store an item
  int pop(); // retrieve an item
}
```

```
class FixedStack implements IntStack {
      //implements fixed size stack
}


Class DynStack implements Intstack{
      //implements dynamic size stack
}
```

Src - Java: The complete Reference, 11th Edition

Interfaces – Dynamic method resolution



```
/* Create an interface variable and
   access stacks through it.
*/
class IFTest3 {
  public static void main(String args[]) {
    IntStack mystack; // create an interface reference variable
    DynStack ds = new DynStack(5);
    FixedStack fs = new FixedStack(8);

    mystack = ds; // load dynamic stack
    // push some numbers onto the stack
    for(int i=0; i<12; i++) mystack.push(i);

    mystack = fs; // load fixed stack
    for(int i=0; i<8; i++) mystack.push(i);

    mystack = ds;
    System.out.println("Values in dynamic stack:");
    for(int i=0; i<12; i++)
       System.out.println(mystack.pop());

    mystack = fs;
    System.out.println("Values in fixed stack:");
    for(int i=0; i<8; i++)
       System.out.println(mystack.pop());
  }
}
```

Src - Java: The complete Reference, 11th Edition

Generics

```
interface IntStack{
        void push(int element); // store the element
        int pop(); //retrieve the item
}
interface CharStack {
        void push(char element); // store the element
        char pop(); //retrieve the item
}
```

Stack operations can be implemented without explicit type

```
interface Stack <T>{
        void push(T element); // store the element
        T pop(); //retrieve the item
}
```

Generics

A class, interface, method that operates on a parametrized type is called *generic*

Generics provide support to write classes, interfaces, methods that work on different types of data in a type-safe manner

e.g., Stack operations can be implemented without explicit type

Java has always provided the ability to support generalized classes through Object reference type

But Generics added type safety

Generics avoids explicit casting

Generics - Example

Types can be declared using generic names:

```
public class Pair<A,B> {
  A first;
  B second;
  public Pair(A a, B b) {                          // constructor
    first = a;
    second = b;
  }
  public A getFirst() { return first; }
  public B getSecond() { return second;}
}
```

They are then instantiated using actual types:

```
Pair<String, Integer> bid = new Pair<>(“XYZ”, 20);
```

Generalized Class using Object

Use Object reference

```
Class ObjectPair(){
  Object first;
  Object second;
  ObjectPair(Object A, Object B){
       first = A;
       second = B;
  }
  public Object getFirst(){return first;}
  public Object getSecond() {return second;}
}
```

Instantiated like any other normal object

```
ObjectPair bid = new ObjectPair(“XYZ”, 20);

String S  = (String) bid.getFirst(); //casting
```

Generalized Class using Object

Use Object reference

```
Class ObjectPair(){
  Object first;
  Object second;
  ObjectPair(Object A, Object B){
       first = A;
       second = B;
  }
  public Object getFirst(){return first;}
  public Object getSecond() {return second;}
}
```

Instantiated like any other normal object

```
ObjectPair bid = new ObjectPair(20, “XYZ”);

String S  = (String) bid.getFirst(); //casting
```

Bounded Generic Type

A formal parameter type can be restricted

For upper bound: use “extended”

For lower bound: use “super”

```
public class NaturalNumber<T extends Integer> {

    private T n;

    public NaturalNumber(T n)  { this.n = n; }

    public boolean isEven() {
        return n.intValue() % 2 == 0;
    }
}

NaturalNumber<Double> nat = new NaturalNumber<>(3.4); // compiler error
```

https://docs.oracle.com/javase/tutorial/extra/generics/wildcards.html

Bounded type parameters are key to generic algorithms

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem)  // compiler error
            ++count;
    return count;
}
```

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

```
public static <T extends Comparable<T>>
       int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

src: https://docs.oracle.com/javase/tutorial/java/generics/

Wildcard Type

Wildcard is used to accept a broader set of class types as argument
Wildcard can be upper bounded/lower bounded/unbounded

```
<? extends T>
<? super T>
```

Ex: An array of elements can be sorted with comparisons and swaps. A generic sorting algorithm can be written that works on any type T, provided T's type hierarchy implements the `Comparable` interface:

```
public static<T extends Comparable<? super T>> void sort(T[ ] arr) { … }
```

https://docs.oracle.com/javase/tutorial/extra/generics/morefun.html

Generic implementation

No new classes created for parameterized types

Replaces all generic type with their bounded type or Object type if unbounded

Insert type cast if necessary, to ensure safety

Generic Restrictions

Some Generics restrictions:

Type parameter cannot be instantiated

```
Class Gen<T>{
      T obj;
      Gen() { obj = new T();}  //illegal
}
```

Cannot create an array of parametric type

```
      T arr = new T[10];
```

Cannot create an array of type specific generic references

```
      Gen<Integer> arr = new Gen<Integer>[10];
```

https://docs.oracle.com/javase/tutorial/java/generics/erasure.html

Comparable interface

```
public interface Comparable<T>
```

Requires the implementation of:

```
        public int compareTo (T x) { ... }
```

"this" object is compared with x and should return

negative value if this < x,
0 if this = x, and,
positive value if this > x

Order imposed by compareTo is known as "natural ordering".

Algorithms like Merge Sort and Binary Search can be implemented on arrays of generic types, as long as they implement this interface in their type hierarchies.

Binary search example

```
/* Implementation of Binary Search algorithm using generics. */
public static<T extends Comparable<? super T>>
  boolean  binarySearch(T[] A, int lo, int hi, T x) {
    while(lo <= hi) {
        int mid = (p+r) >>> 1;   // same as “q = (p+r)/2;”
        int cmp = A[q].compareTo(x);
        if (cmp < 0) {
               lo = mid+1;
        } else if (cmp == 0) {
               return true;
        } else {
               hi = mid-1;
        }
    }
    return false;
}
```

Interface Comparator<T>

```
public interface Comparator<T>
```

Requires the implementation of:

```
        public int compare(T x, T Y) { ... }
```

A comparison function, imposing a total ordering on a collection. Both elements are passed as parameters: compare(x, y).

Returns negative integer of x < y, zero if x = y, and positive integer if x > y.

For consistency, if compare(x, y) returns 0, then x.equals(y) should return true.

Comparable vs Comparator

Comparable	Comparator
Only one ordering (natural)	Many orderings. Implement one comparator for one ordering
Implemented inside the class	Can be implemented outside the class
compareTo(T obj)	Compare(T obj1, T obj2)

Interface Iterator<E>

Iterate over a collection

```
public interface Iterator<T>
```

Methods: `hasNext, next, remove`

Implementations: all collections

Usage:

```
void somefunction(Collection<Item> c) {
   Iterator<Item> it = c.iterator();
   while(it.hasNext()) {
      Item x = it.next();
      ...
   }
}
```

Iterable interface

```
public interface Iterable<T>
```

Iterable: Any type that can be iterated.

Method: `Iterator <T> iterator()`

If a collection class implements the Iterable interface, then “foreach” loop can be used to iterate over its objects:

```
for(Item x: col) { ... }
```

This makes the code more readable.

Java Library

Many data structures are implemented in Java's library:

Lists: `ArrayList`, `LinkedList`, `ArrayDeque`
Sets: `HashSet`, `TreeSet`
Maps: `HashMap`, `TreeMap`
Priority Queues: `PriorityQueue` (binary heaps)
Hashing: `HashMap` (separate chaining)
Balanced binary search trees: `TreeMap` (Red-Black trees)

https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/package-summary.html#CollectionsFramework