Arrays
- a special kind of object (with lots of syntax)
- can declare arrays of any type
- have one instance variable: length
- also have contents indexed with a subscript from 0 … length-1
- can be initialized using \{val_0, val_1, …, val_n\} notation
  - inefficient for large arrays

Array declarations
- Little surprising for C/C++ programmer
- \texttt{int[]} \texttt{A} and \texttt{int [A[]]} have identical semantics
  - declares \texttt{A} to be a variable containing a reference to an array of ints
- \texttt{int[] A[], B;}
  - \texttt{A} is a ref to an array of refs to arrays of ints
  - \texttt{B} is a ref to an array of ints
- None of these allocate an array
- \texttt{A = new int [10];} allocates an array of 10 ints, and makes \texttt{A} a reference to it

Array example
\begin{verbatim}
int[] array1 = {1, 3, 5};
int[][] a = new int[10][3];
// a.length == 10
// a[7].length == 3
a[5][2] = 42;
a[9] = array1;
// a[9][2] == 5
// use of array initializers
int[][] twoD = {{1, 2, 3}, {4, 5}, {6}};
Object [] args = {"one", "two", a};
main(new String [] {"a", "b", "c"});
\end{verbatim}

String
- A class for representing non-mutable strings
- \texttt{string constants} converted to \texttt{String}
- + does string concatenation
- In some contexts objects automatically converted to \texttt{String} type
- Example:
  \begin{verbatim}
  public static void printArray(Object [] a) {
    for (int i = 0; i < a.length; i++)
      System.out.println("a[" + i + "] = " + a[i]);
  }
  \end{verbatim}

Object/memory allocation
- Only way/time an object gets allocated is:
  - by executing \texttt{new}
    - one object per invocation of \texttt{new}
  - by having an array constant (e.g., \texttt{[5, -5, 42]})
  - by having a string constant (e.g., \texttt{"Hello world"})
- Declaring a reference variable does \textit{not} allocate an object
- Allocating an array does not automatically allocate the contents of the array

Allocation (cont.)
- Multi-dimensional array allocation
  - \texttt{int [][] a = new int [10][10];}
  - equivalent to, but faster than:
    \begin{verbatim}
    int [[[a = new int [10][10];
    for(int i=0; i<10; i++) a[i] = new int[10];
    \end{verbatim}
- No explicit deallocate required, nor allowed
Garbage collection

• Java uses garbage collection to find objects that cannot be referenced
  – i.e. do not have any pointers to them
• GC not a major performance bottleneck
  – fast garbage collectors have been implemented
  – on many commercial systems, GC runs on a single processor of a multiprocessor

Other notes

• Forward references resolved automatically
  – can refer to method/variable defined later
• Integer division by zero raises an exception
• Integer overflow drops extra bits
• Floating point errors create special values
  – NaN, POSITIVE_INFINITY, …
• Separate name spaces for methods, classes, variables, …
  – can produce confusing error messages

What’s missing

• preprocessor (#include, #define, …)
• structs and unions
• enumerated types
• bit-fields
• variable-length argument lists
• multiple inheritance (of implementation)
• operator overloading
• templates/ parameterized types
  – GJ (generic Java)

Object-oriented programming in Java

Java Classes

• Each object is an instance of a class
  – an array is an object
• Each class is represented by a class object
  – of type Class
• Each class extends one superclass
  – Object if not specified
  – except class Object, which has no superclass

Classes (cont.)

• Each class has methods and fields/variables
  – variables hold primitive (built-in) values or object references
• Only use ‘.’ to access object fields
  – e.g., x.y(a.b)
• Most methods invoked using C++ virtual method semantics
  – except static, private and final methods
Class modifiers

- **public** – class visible outside package
- **final** – no other class can extend this class
- **abstract** – no instances of this class can be created
  - only instances of extensions of the class

Class details

- Method names can be overloaded
  - method invoked is determined by both its name and the types of the parameters
  - resolved at compile-time, based on compile-time types
- Methods can also be overridden
  - define a method also defined by a superclass
  - arguments and result types must be identical
  - resolved at run-time, based on type of object method is invoked on

Classes and methods

- **this** refers to the object the method is invoked on
- **super** refers to the same object as **this**
  - but used to access methods/variables in superclass
- Methods
  - can be declared in both classes and interfaces
  - only implemented in classes
  - must have a return type
  - except constructors
  - void can be used only as a return type
  - references to objects or arrays can be returned

Instance variable / method modifiers

- Visibility/access
  - **public** – visible everywhere
  - **protected** – visible within same package or in subclass
  - **package** (default) – visible within same package
  - **private** – visible only within this class
  - **static** – a class method or variable

Instance variable modifiers

- **transient** – not stored when object serialized
- **volatile** – don’t assume the variable hasn’t changed since the last time it was accessed
  - might be modified by another thread that doesn’t have a lock on the object
- **final** – can’t be changed, must be initialized in declaration or in constructor
### Method modifiers

- **abstract** – no implementation provided
  - class must be abstract
- **final** – this method cannot be overridden
  - useful for security
  - allows compiler to inline method
- **native** – implemented in another language
- **synchronized**
  - locks object before method is executed
  - lock released after method finishes

### Method arguments

- Only pass-by-value
  - but object parameters are references to heap objects that can be changed
- Only arguments used to distinguish methods
  - not return types
- Syntax same as C/C++

### Overriding

- Overriding
  - methods with same name and argument types in child class override method in parent class
  - you can override/hide instance variables
    - both variables will exist, but don’t do it

```java
class Parent {
    int cost;
    void add(int x) {
        cost += x;
    }
}
class Child extends Parent {
    void add(int x) {
        if (x > 0) cost += x;
    }
}
```

### Overloading

- Methods with the same name, but different parameters (count or types) are overloaded

```java
class Parent {
    int cost;
    void add (int x) {
        cost += x;
    }
}
class Child extends Parent {
    void add (String s) throws NumberFormatException {
        cost += Integer.parseInt(s);
    }
}
```

### Dynamic Method Dispatch

- If you have a ref `a` of type `A` to an object that is actually of type `B` (a subclass of `A`)
  - instance methods invoked on `a` will get the methods for class `B` (like C++ virtual functions)
  - class (static) methods invoked on `a` will get the methods for class `A`
    - invoking class methods on objects strongly discouraged

### Simple Dynamic Dispatch Example

```java
class A {
    String f() {return “A.f()”; }
    static String g() {return “A.g()”; }
}
java B generates:
class B extends A {
    String f() {return “B.f()”; }
    static String g() {return “B.g()”; }

    public static void main(String args[]) {
        A a = new B();
        B b = new B();
        System.out.println(a.f() + a.g() + b.f() + b.g());
    }
}
```