Topic #8:
Arrays and Typing Rules

CSE 413, Autumn 2004

Programming Languages

http://www.cs.washington.edu/education/courses/413/04au/

Readings and References

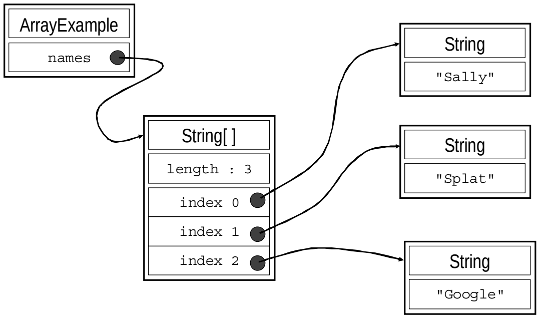
- Reading
  - » Section 12.3 of Sebesta
  - » "Arrays", Java tutorial
    - http://java.sun.com/docs/books/tutorial/java/data/arrays.html

Array Example

```
public class ArraySample {
   public ArraySample() {
      names = new String[3];
      names[0] = "Sally";
      names[1] = "Splat";
      names[2] = "Google";
      for (int i=0; i<names.length; i++) {
         System.out.println("Name "+i+" is "+names[i]);
      }
   }

   String[] names;
}
```

Array Example



Java Array Object

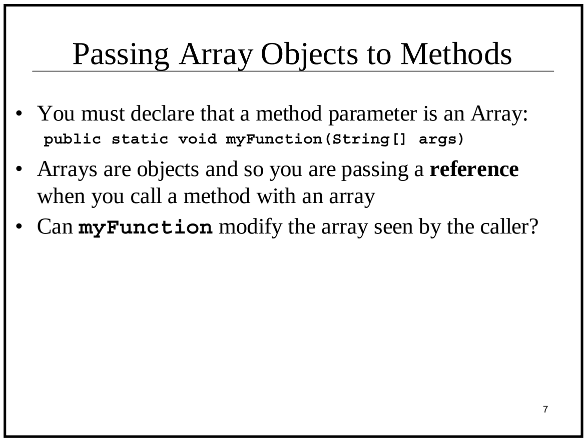
- Arrays are objects! They...
  - » Must be instantiated with **`new`** unless immediately initialized
  - » Can contain **`Object`** references or primitive types
  - » Have class members (length, clone(),…)
  - » Have zero-based indexes
  - » Throw an exception if bounds are exceeded

Array Creation

```
String[] myArray = new String[10];
```

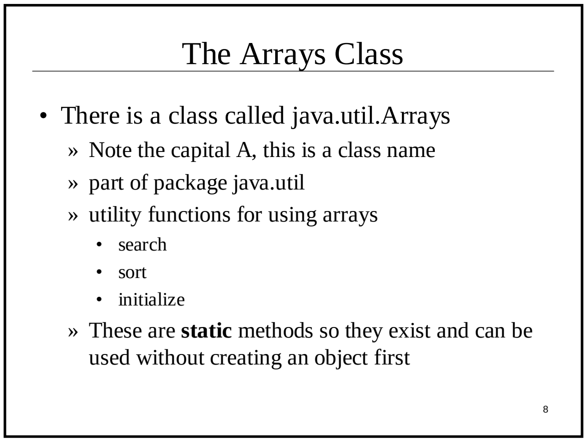
```
String[] myArray = { “Java”,”is”,”cool”};
```

```
boolean okay = doLimitCheck(x,new int[] {1,100});
```



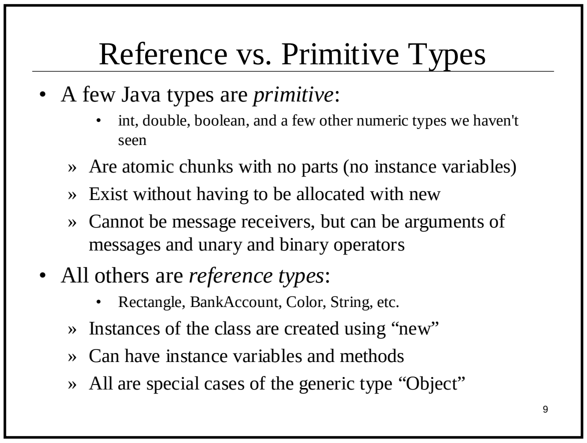
Passing Array Objects to Methods

- You must declare that a method parameter is an Array:
  `public static void myFunction(String[] args)`
- Arrays are objects and so you are passing a **reference** when you call a method with an array
- Can **`myFunction`** modify the array seen by the caller?



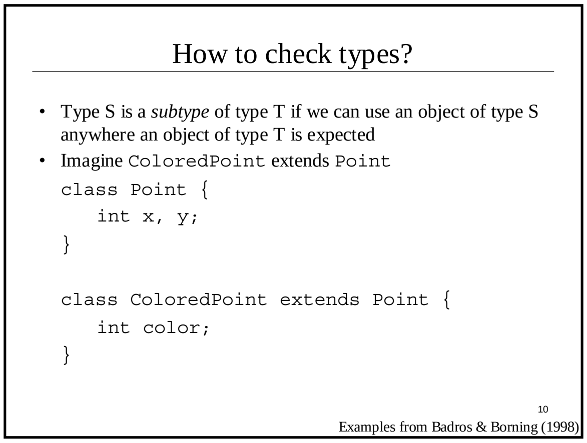
The Arrays Class

- There is a class called java.util.Arrays
  - » Note the capital A, this is a class name
  - » part of package java.util
  - » utility functions for using arrays
    - search
    - sort
    - initialize
  - » These are **static** methods so they exist and can be used without creating an object first



Reference vs. Primitive Types

- A few Java types are *primitive*:
    - int, double, boolean, and a few other numeric types we haven't seen
  - » Are atomic chunks with no parts (no instance variables)
  - » Exist without having to be allocated with new
  - » Cannot be message receivers, but can be arguments of messages and unary and binary operators
- All others are *reference types*:
    - Rectangle, BankAccount, Color, String, etc.
  - » Instances of the class are created using “new”
  - » Can have instance variables and methods
  - » All are special cases of the generic type “Object”



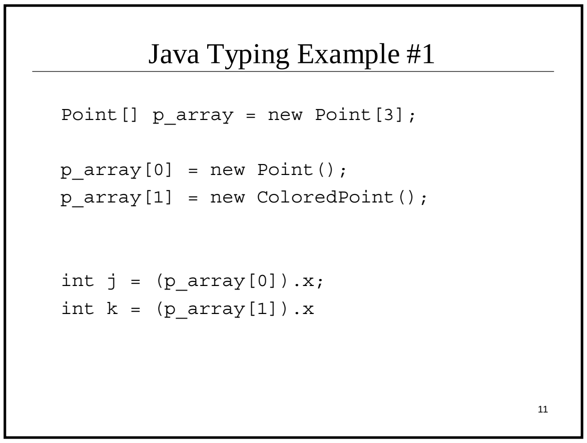
How to check types?

- Type S is a *subtype* of type T if we can use an object of type S anywhere an object of type T is expected
- Imagine `ColoredPoint` extends `Point`

```
class Point {
   int x, y;
}

class ColoredPoint extends Point {
   int color;
}
```

Examples from Badros & Borning (1998)

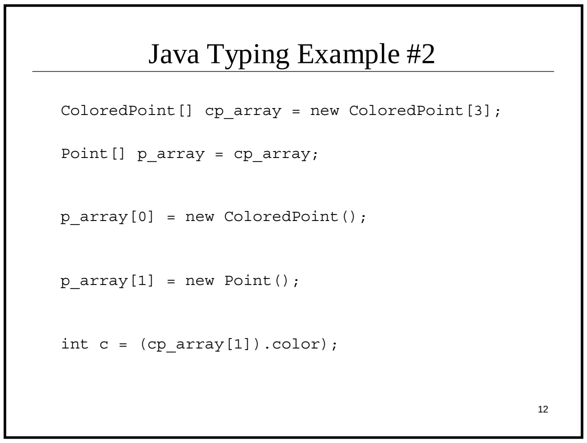


Java Typing Example #1

```
Point[] p_array = new Point[3];

p_array[0] = new Point();
p_array[1] = new ColoredPoint();


int j = (p_array[0]).x;
int k = (p_array[1]).x
```



Java Typing Example #2

```
ColoredPoint[] cp_array = new ColoredPoint[3];

Point[] p_array = cp_array;


p_array[0] = new ColoredPoint();


p_array[1] = new Point();


int c = (cp_array[1]).color);
```

What went wrong?

- Scheme checks types dynamically
- Java checks most types statically
  - » Uses *covariant* rule for arrays – not sound!
  - » So adds a runtime check

General type checking rules (1)

- Type S is a *subtype* of type T if we can use an object of type S anywhere an object of type T is expected
- *Contravariant* rule: S is subtype of T if
  1. S provides all the ops(methods) that T does (maybe more)
  2. For every op in T, corresponding op in S has same number of arguments and results
  3. The types of the results of S's ops are subtypes of the types of corresponding results of T's ops
  4. The types of the arguments of T's ops are subtypes of the types of the corresponding arguments of S's ops
- *Covariant* rule: same as above except
  4. The types of the arguments of S's ops are subtypes of the types of the corresponding arguments of T's ops

General type checking rules (2)

- Type S is a *subtype* of type T if we can use an object of type S anywhere an object of type T is expected
- *Invariant* rule: S is subtype of T if
  1. S provides all the ops(methods) that T does (maybe more)
  2. For every op in T, corresponding op in S has same number of arguments and results
  3. The types of the results of S's ops are the same as the types of corresponding results of T's ops
  4. The types of the arguments of S's ops are the same as the types of the corresponding arguments of T's ops

» **Java uses *contravariant* rule for arrays, *invariant* rule for everything else**

Non-Java Examples (1)

```
Type Color
Type GrayScaleColor (sub type of color)
Type Point:
   int x();
   int y();
Type ColoredPoint:
   int x();
   int y();
   Color mycolor();
Type GrayScalePoint
   int x();
   int y();
   GrayScaleColor mycolor();
```

ColoredPoint subtype of Point??

GrayScalePoint subtype of ColoredPoint??

Non-Java Examples (2)

```
Type Number
Type Integer (sub type of Number)
Type Point:
   Number x();
   Number y();
   void SetDotSize (Integer)
Type ColoredPoint:
   Number x();
   Number y();
   void SetDotSize (Number);
```

ColoredPoint subtype of Point??

Java Examples – what's legal?

```
ColoredPoint extend Point

class PointMaker {
   Point makePoint();
}
class PointEater {
   void eat (Point p);
}
class ColoredPointMaker1 {
   ColoredPoint makePoint()
}
class ColoredPointMaker2 extends PointMaker {
   ColoredPoint makePoint()
}
class ColoredPointEater1 {
   void eat (ColoredPoint p);
}
class ColoredPointEater2 extends PointEater{
   void eat (ColoredPoint p);
}
```

Recall: Java typing problem

```
ColoredPoint[] cp_array = new ColoredPoint[3];

Point[] p_array = cp_array;


p_array[0] = new ColoredPoint();


p_array[1] = new Point();


int c = (cp_array[1]).color);
```

What goes wrong with Arrays?

```
class PointArray {
  Point get(int);
  void set (int, Point);
}
Class ColoredPointArray {
  ColoredPoint get(int);
  void set (int, ColoredPoint);

}
```

Soundness rule:

Are subclasses subtypes in Java?

An Ordered Collection: ArrayList

- ArrayList is a Java class that specializes in representing an ordered collection of things
- We can store *any* kind of object in an ArrayList
  - » myList.add(theDog);
- We can retrieve an object from the ArrayList by specifying its index number
  - » myList.get(0)

ArrayList

- **`ArrayList()`**
  - » This constructor builds an empty list with an initial capacity of 10
- **`int size()`**
- **`boolean add(Object o)`**
- **`Object get(int index)`**

Example

```
import java.util.*; to use ArrayList
ArrayList names = new ArrayList ( );

int numberOfNames = names.size( );

names.add("Billy");
names.add("Susan");
names.add("Frodo");

Object x = names.get(0);
Object y = names.get(1);
```

Using ArrayLists : add

- Adding things

```
names.add("Billy");
```

- The object can be of any class type
  - » String, File, InputStream, …
  - » can’t add “primitive” types like int or double directly

A Problem

- We want to get things out of an ArrayList
- We might write the following:

```
public void printFirstNameString(ArrayList names) {
        String name = names.get(0);
        System.out.println("The first name is " + name);
}
```

- Problem?

Casting

- The pattern is
  - » (<class-name>)<expression>
- For example

  String name = (String)names.get(0);
- Casting an object does ***not*** change the type of the object
- A cast is a promise by the programmer that the object can be used to represent something of the stated type and nothing will go wrong

Miscasting

```
public void printFileList() {
   for (int i=0; i<names.size(); i++) {
        File f = (File)names.get(i);
             System.out.println(f);
   }
}
```

APPENDIX

Array Declaration and Creation

- Array have special type and syntax:

  ***<element type>*[ ]** <*array name*> = **new *<element type>* [*<length>*]**;
- Arrays can only hold elements of the specified type
  - » element type can be int, double, etc.
  - » type can be Object, in which case very similar to ArrayList
- <*length*> is any positive integer expression
- Elements of newly created arrays are initialized
  - » but generally you should provide explicit initialization
- Arrays have an instance variable that stores the length

  <*array name*>`.length`

Array Element Access

- Access an array element using the array name and position: <*array name*> [<*position*>]
- Details:
  - » <*position*> is an integer expression.
  - » Positions count from zero
  - » Type of result is the element type of the array
- Can update an array element by assigning to it:

  <array name> [*<position>*] = *<new element value>* ;

Looping Over Array Contents

- The length attribute makes looping over Array objects easy:

```
for (index=0; index<myArray.length; index++) {
   System.out.println(myArray[index]);
}
```

- The length attribute is a read-only value
  - » You can't change the size of the array after it has been created

Array Summary

- Arrays are the fundamental low-level collection type built in to the Java language.
  - » Also found in essentially all programming languages
- Size fixed when created
- Indexed access to elements
- Used to implement higher-level, richer container types
  - » ArrayList for example
  - » More convenient, less error-prone for users

Using ArrayLists : import

- ArrayList is part of the java.util package
  - » `import java.util.ArrayList;` to use ArrayList
- The import statement tells the Java compiler where to look when it can't find a class definition in the local directory
  - » We tell the compiler to look in package java.util for the definition of ArrayList by putting an import statement at the top of the source code file
  - » Java always looks in package java.lang on its own

Using ArrayLists : size

- Getting the size

```
int numberOfNames = names.size( );
```

- `size()` method returns integer value that caller can use to control looping, check for limits, etc
  - » Design pattern: The object keeps track of relevant information, and can tell the caller when there is a need to know

Using ArrayLists : constructor

- Creating a new ArrayList object
  - `ArrayList names = new ArrayList ( );`
- There are several constructors available
  - » **`ArrayList()`**
    - Construct an empty list with an initial capacity of 10
  - » **`ArrayList(int initialCapacity)`**
    - Construct an empty list with the specified initial capacity
  - » **`ArrayList(Collection c)`**
    - Construct a list containing elements from another collection