CS202 Introduction to Java

- Introduction to Java
  - Philosophy and approach
  - Similarities between C++ and Java
  - Differences between C++ and Java
  - Examine classes, data types, operations, functions and their arguments, arrays, inheritance, and dynamic binding

CS202 Introduction to Java

- Like C++, Java is a hybrid language
  - Which means the syntax is not strictly limited to OOP constructs, although it is assumed that you want to do OOP using Java (e.g., exception handling is not an OOP feature)
  - The benefit is that the initial programming effort should be simpler to learn and use than many other OOP languages
- One of Java's primary goals is to make programming less error prone; for example, Java meets this goal
  - by performing bounds checking
  - by not having explicit pointers
  - by providing automatic garbage collection
- Much of the foundation of C and C++ has been taken as a foundation in Java, with modifications. This is good news for us!
  - On the other hand, unlike C++, Java does not maintain compatibility with the other languages, so you will find larger variations when moving from C or C++ to Java.

CS202 Introduction to Java

- In Java, we treat everything as an object
  - We can have objects of primitive types (like int, float, char) or objects of class types.
  - Objects of primitive types can be created in the same way that we do in C++ (e.g., int object;)
  - Objects of class types cannot be created this way.
    - First, we must create identifiers for objects that we desire – these are actually references to objects
    - Then, we must allocate memory on the heap for instances
  - So, when we say: List obj; we have created only a reference not an object. If you want to send a message to obj (i.e., call a member function), you will get an error because obj isn't actually pointing to anything: List obj = new List();
  - So, for a string object we could say:
    - String s = "CS202!"; or
    - String s = new String("CS202!");

CS202 Introduction to Java

- When we create a reference, we want to connect it to a new object
- String s = new String (“CS202!”); //or
- List obj = new List(); //default constructor…
  - We do this with the new keyword
  - This allocates memory for a new string and provides an initial value for the character string
  - And, like in C++ this causes the constructor for the class type to be implicitly invoked
  - New places the objects on the heap (which is not as flexible as allowing objects to be allocoated on the stack)
  - It is not possible to request local objects of a class or to pass objects of a class by value to a function (or as the return value). This is because we are always working with a reference to the object – which is what gets passed (by value)

Classes in Java

- Everything we do in Java is part of a class
  - This means that none of our functions can be “global” like they can be in C or C++
- Classes in Java specify types, as they do in C++ and allow us to create abstractions
  - Classes must be specified as public, or not.
  - Only public classes are available for the outside world to create objects of
  - If the keyword public doesn’t precede a class, then it is “friendly” and only classes from within this file or package (a group of files) can create objects Every package (or file to begin with) has a public class
- Inside of a class, Java supports public, protected, and private access (or nothing – which means “friendly” access)
  - But unlike C++, it requires that they be specified in front of each member rather than specifying categories!
  - This means that everything has some kind of access specified for it.

Class Access Visibility

- Unlike C++, a class can be specified as public, or not. A public class within a library specifies which class(es) are available to a client programmer
  - The public keyword just has to be placed somewhere before the opening brace ({) of the class body
  - There can be only one public class per compilation unit
  - They must be the same name as the file
  - Without the public qualifier, a class becomes “friendly”, available to the other classes in the library to which it belongs
  - Classes cannot be private or protected.
  - If you don’t want anyone access to a class, then make the constructors private!

Members of a Class

- Like C++, classes in Java can have member functions (called methods) and data members
  - These look the same as in C++ except for the access control specifier and
  - The implementation of the member functions is provided (in most cases) in the class itself – not elsewhere
- Each member can be specified as public, private, or protected.
- If a member has no access control (public, private or protected), they are treated as “friendly”
  - the default means “friendly”: all other classes in the current package have access but all classes outside of this package are private
  - This allows us to make data members and member functions semi “global” – the scope is somewhat broader than C++ static global but restricted from actually be global in nature.
  - It allows us to create a library (a package) and allow related classes to access members directly
  - This helps us to organize classes together in a meaningful way

Class Access Specifiers

- What is the meaning in Java of public, private, and protected?
- As is expected, public means that the member following is public to everyone using this library
- Private is not available – only other methods within the class can access that member
  - Helper methods should be private
- Protected members are available within this class and a derived class (same as C++)
- Recommendation: limit your use of the default friendly access
  - (Please note, a derived class may not be able to access its parent's "friendly" members if they are not in the same package! This is because it is possible to inherit from a class that is not in the same package.)

Member Accessibility

External access	public	protected	(default) package	private
Same package	yes	yes	yes	no
Derived class in another package	yes	yes (inheritance only)	no	no
User code	yes	no	no	no

Similarities to C++

- You will find much of the syntax similar to C++
  - (primitive types, compound blocks, loops (all), built-in operators (most), switch, if-else),
  - static data members,
  - casting for primitive types,
  - scope of your loop control variables in a for,
  - allowing definitions of a variable to occur where you need them,
  - use of the ++ and – operators,
  - Break and continue

Similarities to C++

- Other similarities between Java and C++:
  - Two types of comments (// and /* */)
  - Support for function overloading (unlike C)
  - Static member functions (equivalent to C's global functions)
    - Global functions are not permitted in Java
    - They don't have a this pointer
    - They can't call a non-static member function without an object of the class
    - Don't overuse them if you are doing OOP!

CS202 Introduction to Java

- Minor Differences
  - You cannot define the same named variable in different – inner vs outer blocks (unlike C++ allows identifiers in an inner block to hide those in an outer scope
  - Primitive types in Java are guaranteed to have an initial value (i.e., not garbage!)
  - Java determines the size of each primitive type (they don't change from one machine architecture to another – unlike C and C++)
  - All numeric types are signed – they do not support the unsigned type.
  - No semicolon is required at the end of a class definition

CS202 Introduction to Java

- Minor Differences
  - No const – instead we use “final” to represent memory that cannot be changed
    - final int I = 10; //a constant value
    - final list object = new list(); //reference is constant – so it can’t reference another object! However, the object itself can be modified
  - This means that class objects in fact can’t be constant!
  - Final doesn’t require that the value of the variable/object be known at compile time final list obj; //says it is a “blank” final reference
  - Blank finals must be initialized in the constructor where the blank final is a member
  - Arguments can also be “final” by placing the keyword in the argument list – which means the method cannot change the argument reference

CS202 Introduction to Java

- Minor Differences
  - Although data members are initialized automatically (and so are arrays, variables of primitive types used in a function (i.e., local variables) are not automatically initialized (e.g., int var;)
  - You are responsible for assigning an appropriate value to your local variables
  - If you forget, you will get an error message indicating that the variable may not be initialized.
  - Also…ints are not bools in Java, you can't use an int as part of a conditional expression like we are used to. So saying (while (x=y)) can't happen!
    - Because the result of the expression is not a boolean and the compiler expects a boolean and won't convert from an int
    - So, unlike C++ you will get an error if you make this mistake!

CS202 Introduction to Java

- Minor Differences
  - Remember the comma operator in C++?
    - In Java it can only be used in for loops to allow for multiple increment steps
  - There is no operator overloading
    - Which means you cannot compare strings with > >= etc.
    - You cannot assign objects to do a complete copy (=)
    - You cannot read and write using >> or <<
  - You cannot cast class types!
    - To convert you must use special methods (i.e., function calls)
  - But, you can assign data members values – directly:

```
class list {
        int I = 100;
        video v = new video();
```

Java Identifiers

- **A Java identifier must start with a letter or underscore or dollar sign, and be followed by zero or more letters (A-Z, a-z), digits (0-9), underscores, or dollar signs.**

VALID

age_of_dog **taxRateY2K**

HourlyEmployee **ageOfDog**

NOT VALID (Why?)

age# **2000TaxRate** **Age-Of-Dog**

What is an Identifier?

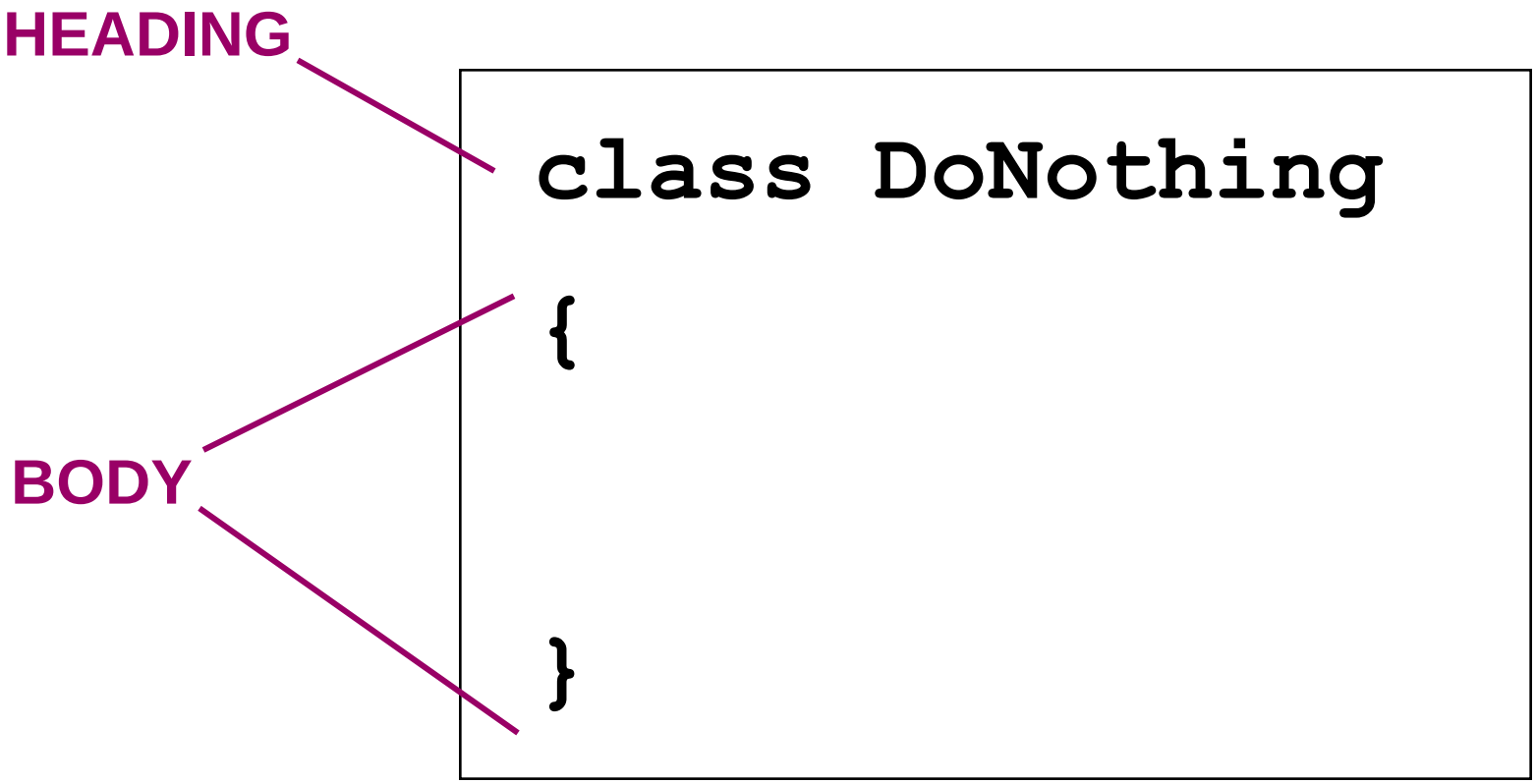
- An identifier names a class, a method (subprogram), a field (a variable or a named constant), or a package in a Java application
- Java is a case-sensitive language; uppercase and lowercase letters are different
- Using meaningful identifiers is a good programming practice

51 Java Reserved Words

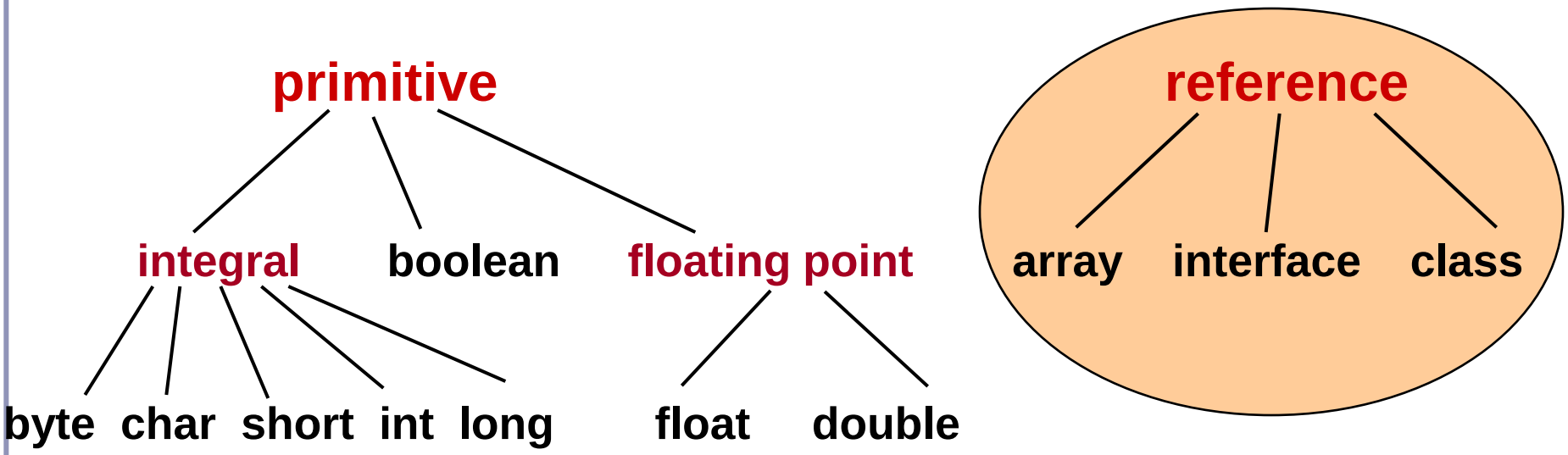
abstract	boolean	break	byte	case
catch	char	class	const	continue
default	do	double	else	extends
false	final	finally	float	for
goto	if	implements	import	instanceof
int	interface	long	native	new
null	package	private	protected	public
return	short	static	strictfp	super
switch	synchronized	this	throw	throws
transient	true	try	void	volatile
while				

Reserved words cannot be used as identifiers.

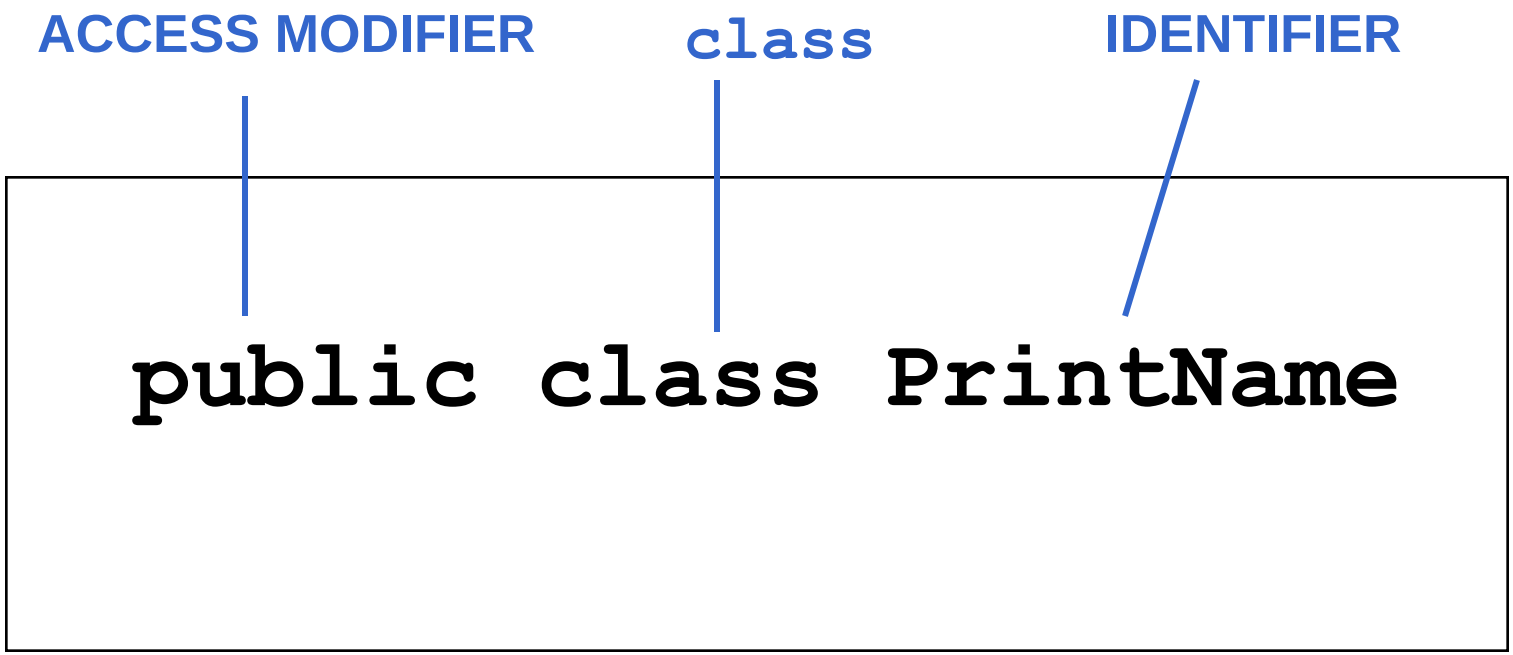
Simplest Java class



Java Data Types



What's in a class heading?



Syntax for Declarations

Variable Declaration

```
Modifiers  TypeName  Identifier , Identifier  . . . ;
```

Constant Declaration

```
Modifiers  final TypeName  Identifier = LiteralValue;
```

Operators?

- Almost all operators work **only** with primitives (not class types)
  - And the operators are those that you know (except there is no sizeof operator)
- =, == and != work on all objects of any type (even class types!!!!)
  - But, if you use them with a reference to an object – you are just manipulating the references.
  - = causes two object references to point to the same object (feels like shallow copy!)
  - == and != compares two references to see if they are pointing to the same object (or not)!
  - And, since there is no operator overloading – we can't change this to do a deep copy!
  - This is because Java allows us to use references truly as aliases. You can cause a deep copy to happen simply (?) by coping each of the members directly that are part of a class or calling a member function to do this
- The String class also supports + and +=

Operators? Equals() method

- If you want to do a deep comparison
  - you “must” (can?) call a method (equals()) that exists for all objects of class type.
  - Of course, the default behavior of equals() is to just compare the references
  - So you “must” (should) override the equals() so that it actually compares the memory contents of the object
  - I recommend you always override this!

What about Arrays?

- Arrays are available in Java,
  - But unlike C and C++, one of Java's primary goals is safety.
  - So, a Java array is guaranteed to be initialized and it cannot be accessed outside of its range
  - Range checking requires a small amount of memory overhead on each array as well as index verification at run time.
  - And, as shown on the previous slide, argument passing with arrays are considerably different (look where the [] go!)

Arrays of Objects

- When you create an array of objects
  - You are really creating an array of references
  - Which are automatically initialized to null
  - Java interprets a null as being a reference which isn't pointing to an object
  - You must assign an object to each reference before you use it!
  - And, if you try to use a reference while it is still null, you will get a run-time error reported
  - Plus, Java provides for range checking – so that arrays cannot be accessed outside of range

Arrays are defined…

- int [] array_name; int array_name [];
- You don't specify the size of an array because no space is allocated for the elements at this point
- All we have is a reference to an array
  - (like in C++ where the name of the array is the starting address of the first element, now in Java the name of the array is a reference)
- To allocate memory we must specify an initialization expression (which unlike C++ can happen anywhere in your code
  - int [] array_name = {1,2,3,4,5}; //starting with element zero
  - A reference can then be used to also access this array:
  - int [] reference;
  - reference = array_name;
- We can also allocate arrays on the heap
  - reference = new int [size];
- All arrays have an implicit member that specifies how many elements there are (its length)

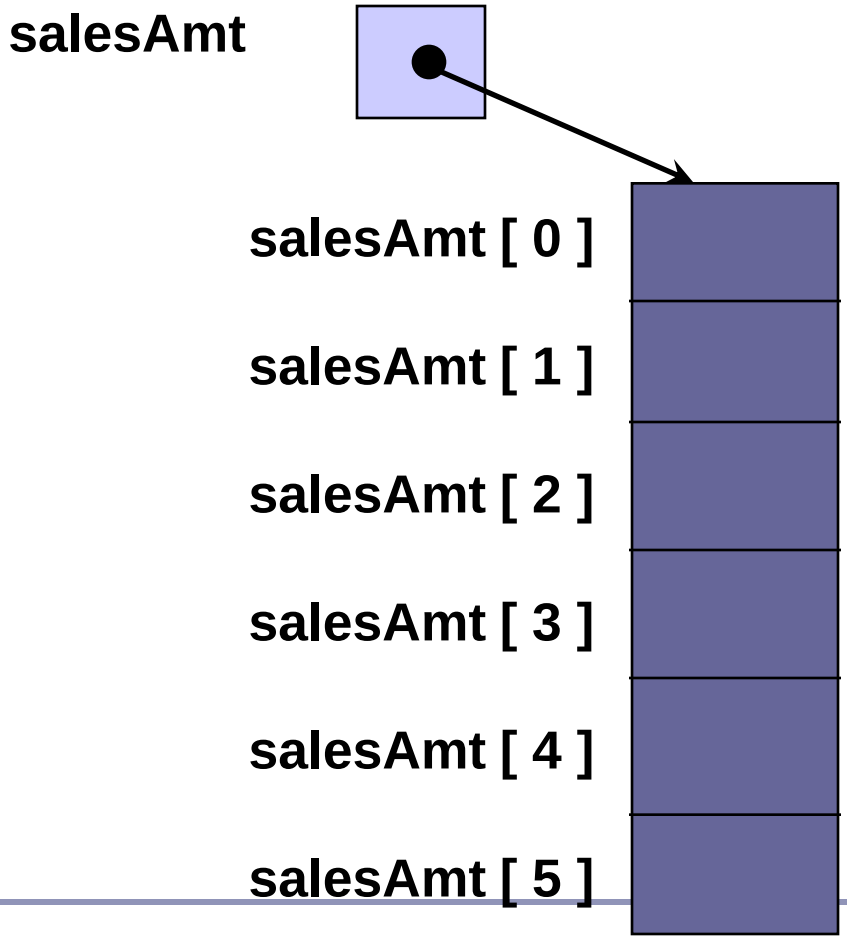
Arrays of class type…

- All arrays of class types must be defined using new (with an exception of the String class….)
  - list [] array = new list[size];
- But, such arrays are actually arrays of references to our objects – not instances!
  - (like an array of pointers in C++)
- If you forget to allocate objects for the elements, you will get an exception
- List [] array = new list[] {new list(1), new list (2), new list(3)};
- Or, do this explicitly with a loop
- Unlike C and C++, Java allows the return type of functions to be an array

Arrays

- Arrays are data structures consisting of related data items all of the same type
- An array type is a reference type; contiguous memory locations are allocated for an array, beginning at the base address
- The base address is stored in the array variable
- A particular element in the array is accessed by using the array name together with the position of the desired element in square brackets; the position is called the index or subscript

```
double[] salesAmt;
salesAmt = new double[6];
```



Array Definitions

- Array A collection of homogenous elements, given a single name
- Length A variable associated with the array that contains the number of locations allocated to the array
- Subscript (or index) A variable or constant used to access a position in the array: The first array element always has subscript 0, the second has subscript 1, and the last has subscript length-1
- When allocated, the elements are automatically initialized to the default value of the data type: 0 for primitive numeric types, `false` for `boolean` types, or `null` for references types.

Another Example

- Declare and instantiate an array called `temps` to hold 5 individual double values.

number of elements in the array

```
double[ ]  temps = new double[ 5 ];

           // declares and  allocates memory
```

temps[0] temps[1] temps[2] temps[3] temps[4]

indexes or subscripts

Declaring and Allocating an Array

- Operator `new` is used to allocate the specified number of memory locations needed for array DataType

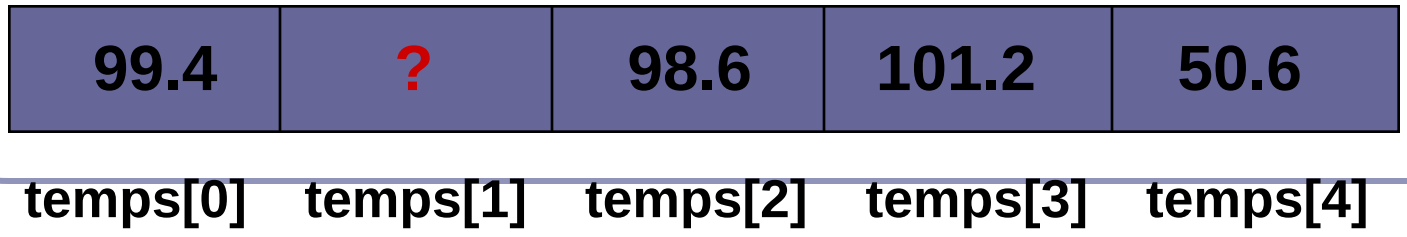
SYNTAX FORMS

```
DataType[ ]  ArrayName;                          // declares array
ArrayName =  new DataType [ IntExpression ];     // allocates array
```

```
DataType[ ]  ArrayName  =  new DataType [ IntExpression ];
```

Assigning values to array elements

```
double[] temps = new double[5];       // Creates array
int m = 4;
temps[2] = 98.6;
temps[3] = 101.2;
temps[0] = 99.4;
temps[m] = temps[3] / 2.0;
temps[1] = temps[3] - 1.2;
 // What value is assigned?
```



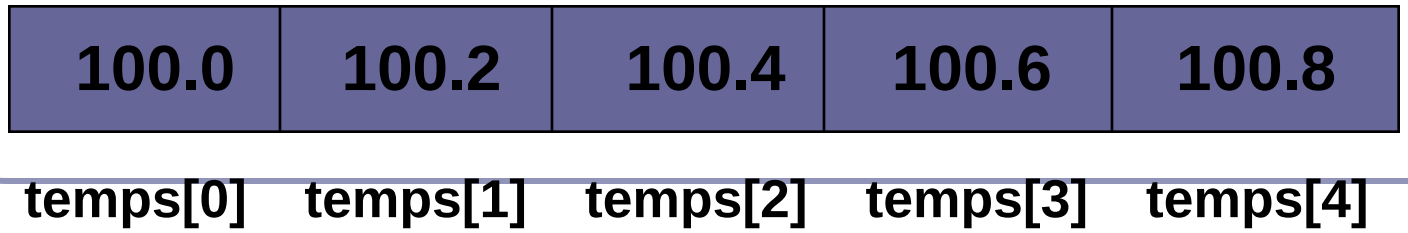
What values are assigned?

```
double[] temps = new double[5]; // Allocates
   array
int m;

for (m = 0; m < temps.length; m++)
   temps[m] = 100.0 + m * 0.2;
```

temps[0] temps[1] temps[2] temps[3] temps[4]

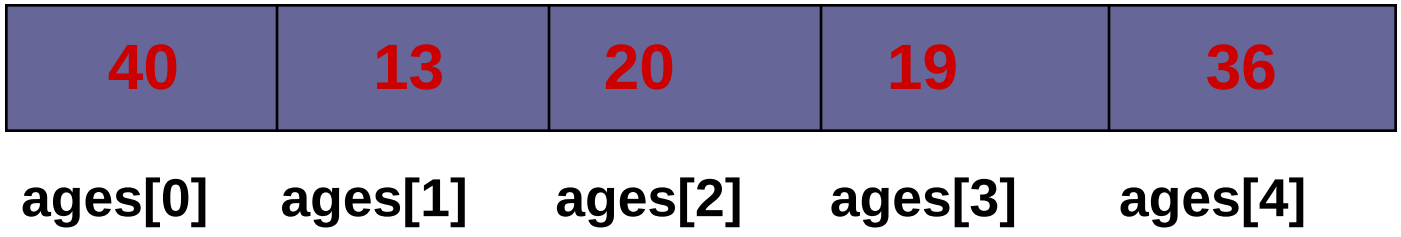
Now what values are printed?

```
final int ARRAY_SIZE = 5;                    // Named constant
double[] temps;
temps = new double[ARRAY_SIZE];
int m;
 . . . . .
for (m = temps.length-1; m >= 0; m--)
  System.out.println("temps[" + m + "] = " + temps[m]);
```



Initializer List

```
int[] ages = {40, 13, 20, 19, 36};

for (int i = 0; i < ages.length; i++)
  System.out.println("ages[" + i + "] = " +
    ages[i]);
```



Passing Arrays as Arguments

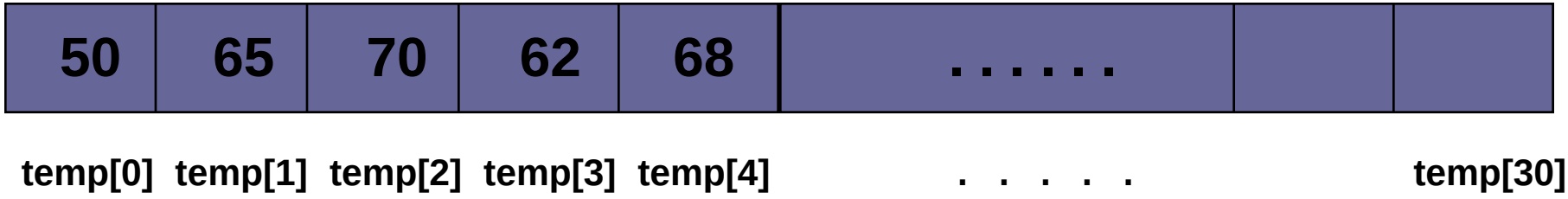
- In Java an array is a reference type. The address of the first item in the array (the base address) is passed to a method with an array parameter
- The name of the array is a reference variable that contains the base address of the array elements
- The array name dot length returns the number of locations allocated

Passing an Array as Arguments

```
public static double average(int[] grades)
// Calculates and returns the average grade in
   an
// array of grades.
// Assumption: All array slots have valid data.
{
   int total = 0;
   for (int i = 0; i < grades.length; i++)
      total = total + grades[i];
   return  (double) total / (double)
   grades.length;
}
```

Memory allocated for array

```
int[] temps = new int[31];
// Array  holds 31 temperatures
```

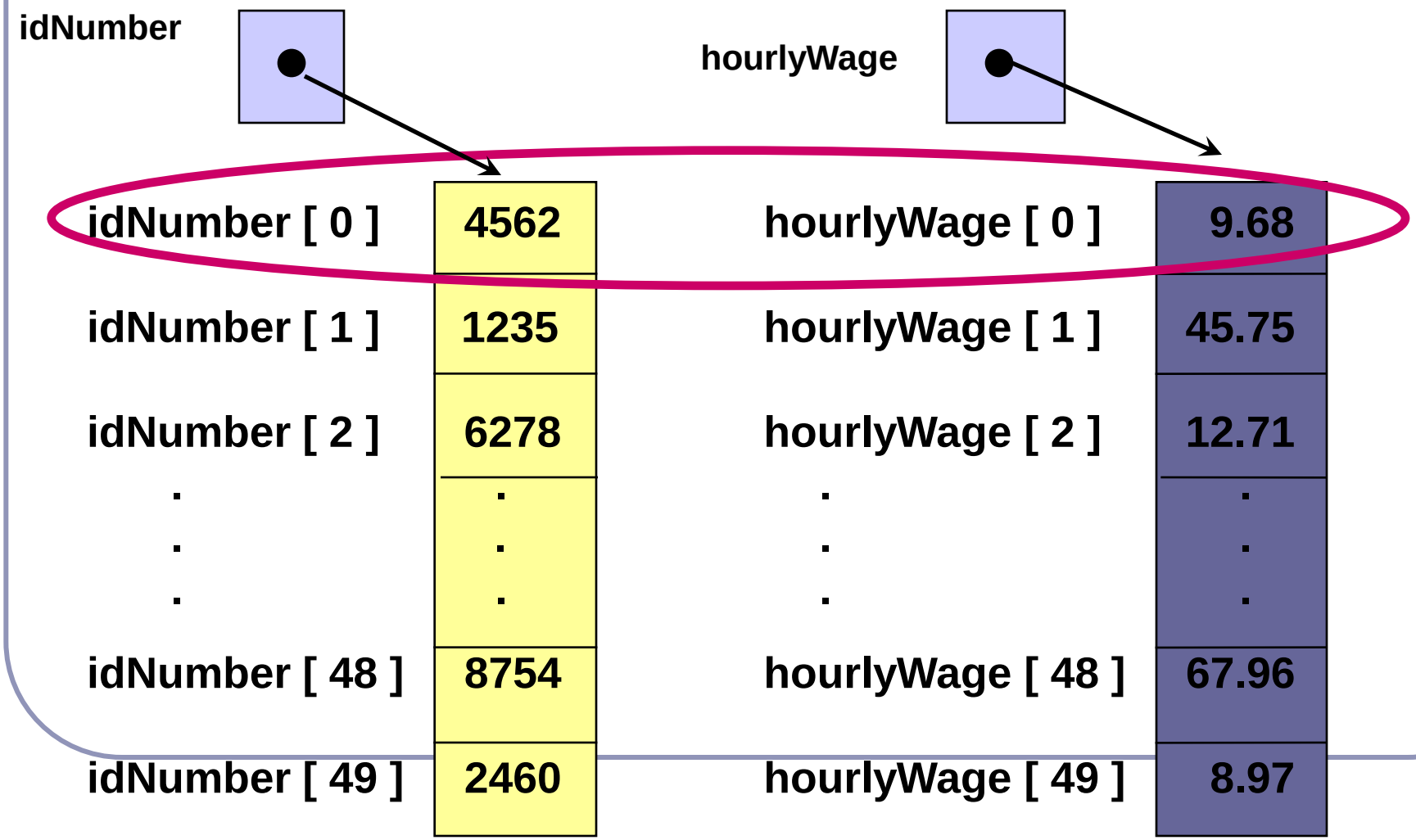


Parallel arrays

- **Parallel arrays** **Two or more arrays that have the same index range, and whose elements contain related information, possibly of different data types**

```
final int SIZE = 50;
int[] idNumber = new int[SIZE];
float[] hourlyWage = new float[SIZE];
```

```
final  int  SIZE  = 50 ;
int [ ]  idNumber  = new  int  [ SIZE ] ;          // parallel arrays
hold
float [ ]  hourlyWage = new  float [ SIZE ] ;    // related
information
```



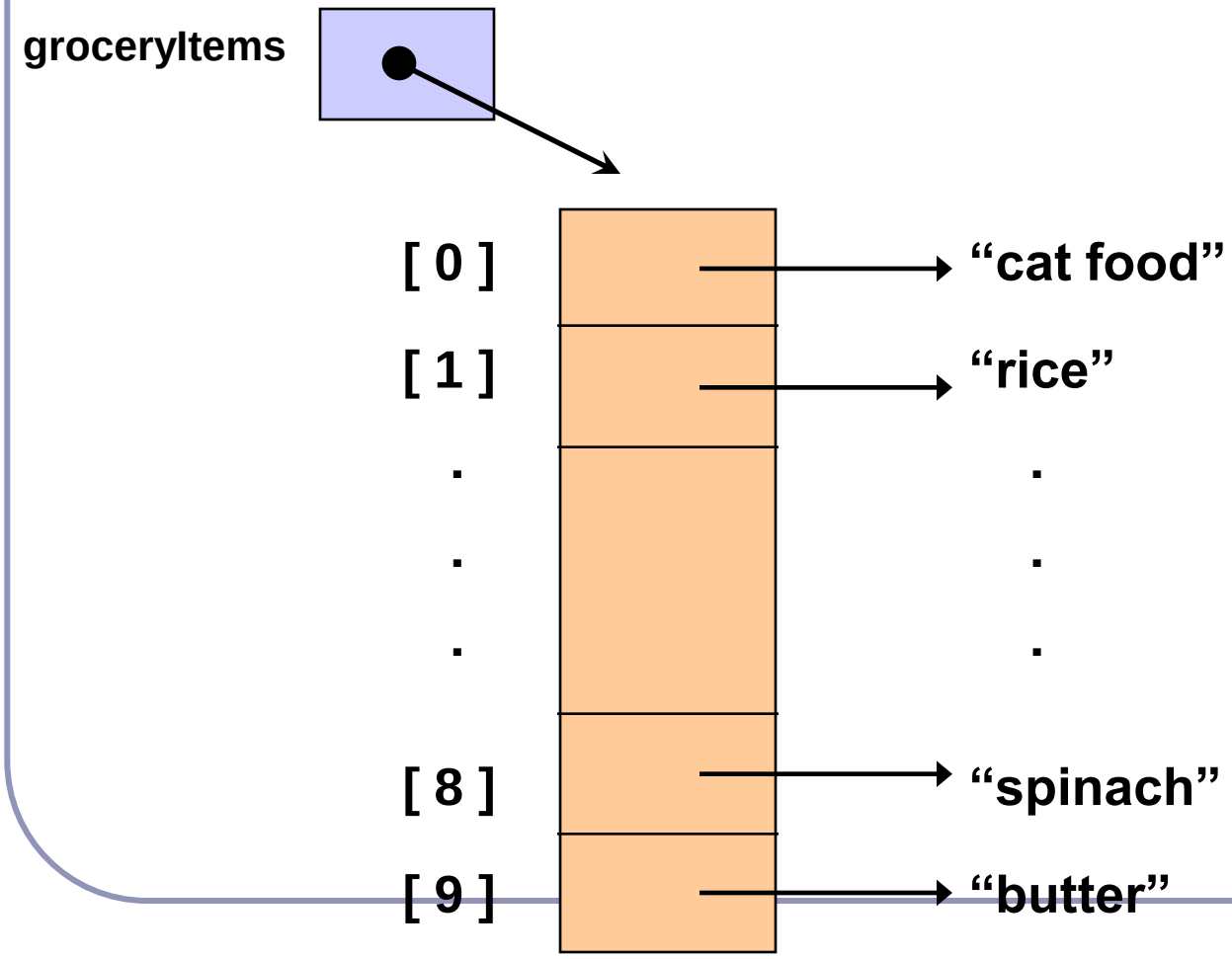
Partial Array Processing

- length is the number of slots assigned to the array
- *What if the array doesn't have valid data in each of these slots?*
- Keep a counter of how many slots have valid data and use this counter when processing the array

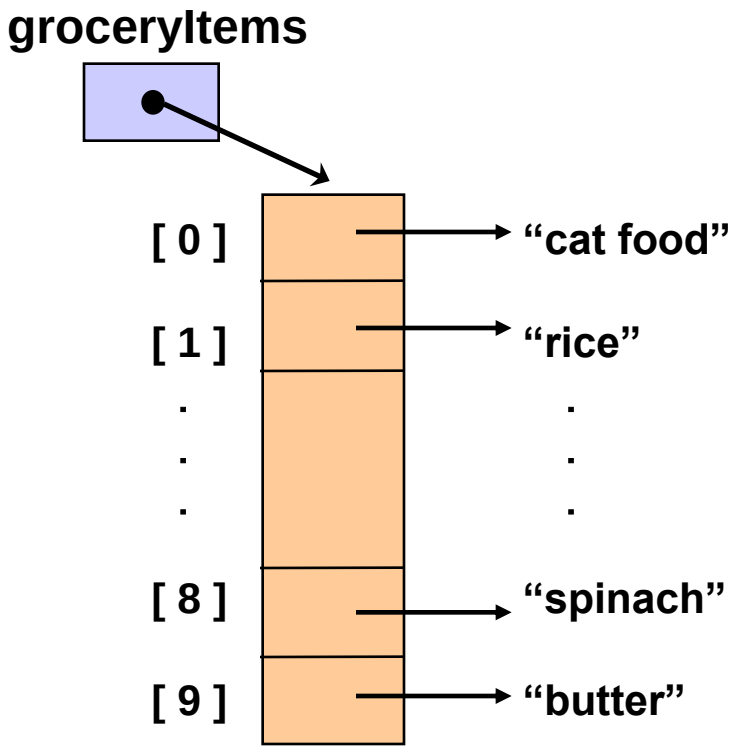
More about Array Indexes

- Array indexes can be any integral expression of type `char`, `short`, `byte`, or `int`
- It is the programmer's responsibility to make sure that an array index does not go out of bounds. The index must be within the range 0 through the array's length minus 1
- Using an index value outside this range throws an `ArrayIndexOutOfBoundsException`; prevent this error by using public instance variable `length`

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

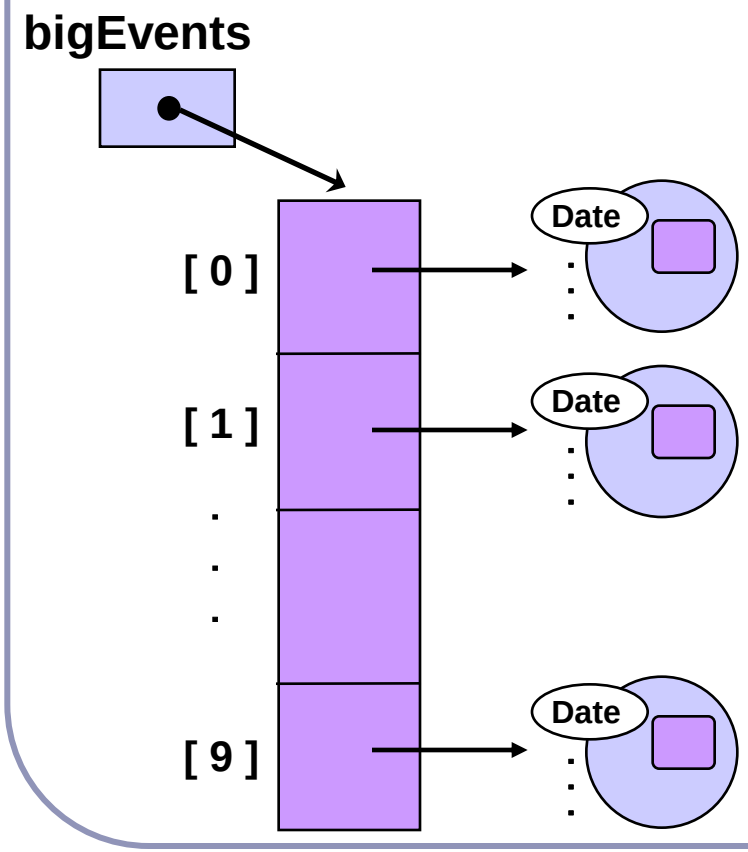


```
String[] groceryItems = new String[10];
```



Expression	Class/Type
groceryItems	Array
groceryItems[0]	String
groceryItems[0].charAt(0)	char

```
Date[] bigEvents = new Date[10];
```



Expression	Class/Type
bigEvents	Array
bigEvents[0]	Date
bigEvents[0].month	String
bigEvents[0].day	int
bigEvents[0].year	int
bigEvents[0].month.charAt(0)	char

Garbage Collection and Objects

- Another difference with Java is that
  - You never need to destroy an object!!!!!!!
  - Java simplifies the need to manage the lifetime of our objects and manages the cleanup work implicitly!
  - When you create an object using new, it actually exists past the end of the block in which it was defined (although the reference to it ends)
  - This is because Java has a garbage collector

Garbage Collection and Objects

- Remember the problems of returning local objects in C++ where the lifetime has ended?
  - We don't have this type of problem in Java because objects created with new exist for as long as we need them and we don't have to worry about destroying them
  - Java has a garbage collector, which looks at all of the objects created with new and determines which ones are not being referenced anymore – then it can release the memory for those objects at that point so the memory can be used for new objects.
  - Please keep in mind that although the garbage collector can release the memory when no more references point to the memory, it may not if the memory is not needed elsewhere
  - On the other hand, this means that you never need to worry about reclaiming memory yourself
  - Simply create objects, and when you no longer need them they will go away by themselves whenever necessary
  - There are no memory leaks!

Garbage Collectors: Efficiency?

- Why doesn't C++ have garbage collection?
  - There is price to it: run time overhead
- C++ allows objects of a class to be created on the stack, not available in Java for class objects
  - These are automatically cleaned up
  - Providing the most efficient way of allocating storage
- Allocating memory on the heap using new is more expensive
  - We have done it in 163/202 to get experience
  - But, in fact it shouldn't be exclusively used!
  - And, it requires that we allocate and deallocate our memory in C++
  - But, in Java, this memory need not be deallocated
- The main issue with garbage collection is that you never really known when it is going to start up or how long it will take
  - This means there is an inconsistency in the rate of execution
  - Which can be important for some real-time software problems

Clarifying References

- Let's clarify our creation of objects in Java
  - Instances of a primitive type (int, float, etc.) are not references and don't need to be created using new
  - In fact, we can't create them using new (except for the case of an array of primitive types)
  - When we create an object of a user defined type (i.e., a class type) we are in fact creating references which means new must be used to actually allocate memory for the instance of the type expected
  - We can then use references to an object using the (.) between the object reference and the member name:
    - Objectreference.member

Types in Java

- Just like C++, we use the keyword class to mean that we are creating a new type
  - class Mytype {…} creates a new data type
- And, creating objects of this type is done using new:
  - Mytype object = new Mytype();
- Like C++, our classes have data members (fields) and member functions (methods)
- Just like objects outside of a class, data members can be of a primitive type or can be references to another user defined class type (requiring the use of new to actually create an instance of them)
- Unlike C++, primitive types can be initialized directly at the point of definition in the class and references can be initialized to connect to objects in the class as well

Primitive Wrapper Classes

- To get a primitive type on the heap,
  - you have to use a wrapper class:
  - (Boolean, Character, Byte, Short, Integer, Long, Float, Double, Void)
    - Character Reference = new Character(‘z’);
  - But, since there is no operator overloading
    - We must use methods instead of operators when working with them

Three Categories of Data

- **Instance data** is the internal representation of a specific object. It records the object's state.
- **Class data** is accessible to all objects of a class.
- **Local data** is specific to a given call of a method.

Categories of Responsibilities

- **Constructor** An operation that creates a new instance of a class
- **Copy constructor** An operation that creates a new instance by copying an existing instance, possibly altering its state in the process
- **Transformer** An operation that changes the state of an object
- **Observer** An operation that allows us to observe the state of an object without changing it
- **Iterator** An operation that allows us to process all the components of an object one at a time

Instance Data

Instance data is the internal representation of a specific object.

```
public class Name
{
// Instance variables
String first;
String middle;
String last;
    . . .
}
```

Class Data

- **Class data is accessible to all objects of a class.**
- **Fields declared as static belong to the class rather than to a specific instance.**

```
public class Name
{
  // Class constant
  static final String PUNCT = ", ";
      . . .
}
```

Local Data

- **Local data is specific to a given call of a method.**
- **Memory for this data is allocated when the method is called and deallocated when the method returns.**

```
public int compareTo(Name otherName)
{
   int result;     // Local variable
   . . .
  return  result;
}
```

Functions (ahhh Methods!)

- Functions in Java are called methods (OOP terminology) and can only be defined as part of a class
  - Luckily, they have the same format we are used to – with return types, argument lists, bodies and return abilities
- Formal arguments have a data type followed by the argument's identifier
  - Unlike C++, you do not get to select whether they are passed by value or by reference.
  - Technically, you could argue that everything is passed by value.
  - Primitive types are passed by value on the stack (you have no choice) and
  - Object references are also passed by value on the stack (keep in mind this is the reference not the object), which "feels" like pass by reference
  - Again, for user defined types, they are actually references automatically (no – you don't put the & or the * in Java in your argument lists!)

Functions (ahhh Methods!)

- For example:

```
int my_func(String s) {
      return s.length();
      }
```

  - The length method returns the number of characters in the string
  - s is actually a reference to the calling routine's string object
  - void is available in Java as it is in C++ to return nothing from the function
  - While object references are placed on the stack when a function is called – the objects to which they refer are not (never). There is no support of a "pass by value" concept with objects of a class.
  - Therefore, we will never perform a deep copy as part of a function call

Method Declaration Syntax

Method Declaration

```
Modifiers  void  Identifier  (ParameterList)
{
        Statement
            . . .
}
```

Methods

- **Method heading and block**

```
void setName(String arg1, String arg2)
{
  first = arg1;
  second = arg2;
}
```

- **Method call (invocation)**

```
Name myName;

myName.setName("Nell", "Dale");
```

Some Definitions

- **Instance field** A field that exists in ever instance of a class

```
String first;
String second;
```

- **Instances method** A method that exists in every instance of a class

```
void setName(String arg1, String arg2);
myName.setName("Chip", "Weems");
String yourName;
yourName.setName("Mark", "Headington");
```

More Definitions

- **Class method** A method that belongs to a class rather than it object instances; has modifier **static**

```
Date.setDefaultFormat(Date.MONTH_DAY_YEAR);
```

- **Class field** A field that belongs to a class rather than its object instances; has modifier **static**

Will cover class fields in later chapters

More Definitions

- **Constructor method** Special method with the same name as the class that is used with `new` when a class is instantiated

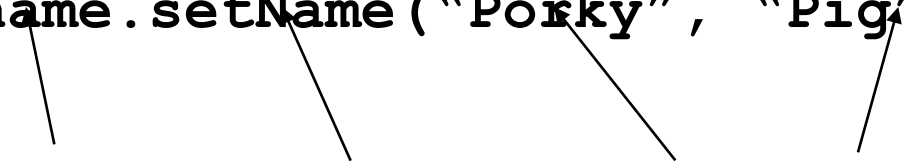
```
public Name(String frst, String lst)
{
  first = frst;
  last = lst;
}
Name name;
name = new Name("John", "Dewey");
```

Note: argument cannot be the same as field

Void Methods

- **Void method** Does not return a value

```
System.out.print("Hello");
System.out.println("Good bye");
name.setName("Porky", "Pig");
```



object method arguments

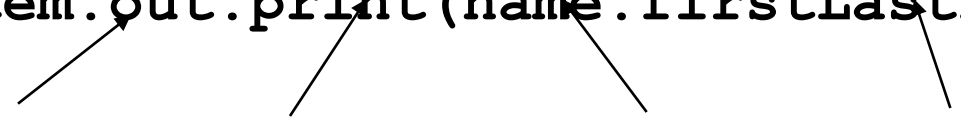
Value-Returning Methods

- **Value-returning method** **Returns a value to the calling program**

```
String first;  String last;
Name name;
System.out.print("Enter first name: ");
first = inData.readLine();
System.out.print("Enter last name: ");
last = inData.readLine();
name.setName(first, last);
```

Value-returning example

```
public String firstLastFormat()
{
  return first + " " + last;
}

System.out.print(name.firstLastFormat());
```



object method object method

Argument to `print` method is string returned from `firstLastFormat` method

The This “Reference”

- When memory for an object is allocated, a reference to that object is created and called the “this” reference
- Like C++, it is the first implicit argument to each method
- Unlike C++, it is not a pointer but rather a reference!
  - list func() { return this; }
  - Which means we do not need to dereference it
  - It allows member concatenation:
  - Obj.func().func().func(); //etc.

Constructors

- Like C++, constructors are implicitly invoked
- They allow us to initialize data members to other values than their zero equivalent
- Note, unlike C++ data members are automatically initialized prior to a constructor invokation (to their zero equivalent) --- even if you provide a constructor
- The default constructor has no arguments
- If you write a constructor with arguments, then the default constructor is not provided automatically and you cannot create objects without arguments specified

```
class list {
        list () {  ///blablabla}
        list (int arg) {  //blablabla }
//we create objects via;
        list l = new list(10);          //uses the int arg version
```

- Yes, you can overload multiple constructors just so the argument lists are unique

Differences with Constructors

- When you write multiple constructors, sometimes we like to have the contructors call another function to actually get the work done (to minimize duplication of code)
- In C++ we do this by writing named member functions
- In Java we do this by having one constructor call another constructor with a special usage of the this pointer!
  - This can only happen once within a constructor
  - It must be the first thing a constructor does

```
list (int i) { //first constructor which does the real work}
list (int i, int j) {this(i);  //calls the constructor with an int }
```

No Destructors!?

- Since Java provides garbage collection
  - There are no destructors
  - But…have you ever had a destructor do something other than memory deallocation?
  - If you need this – you must write a named function and call it explicitly!! (maybe called : void cleanup()?)
- If for some reason you do need some kind of garbage collection done that the garbage collector doesn't know about (like C or C++ memory allocation is being used-not recommended!)
  - You can write a method called "finalize()" which the garbage collector will implicitly call if it is provided prior to releasing memory –
  - and then on the next garbage collection pass it will reclaim the object's memory

finalize() is not a destructor!

- But! This is not a destructor.
- Java objects do not always get garbage collected –
- The garbage collector is only run after all references to an object have been released and memory is insufficient (or running low). It may just automatically return the memory to the operator system after execution!
  - So, use finalize() for releasing memory that the garbage collector cannot predict, but you may need to explicitly cause the garbage collector to be exeucted: System.gc()
- Bottom line, finalize() cannot be relied upon.
- Even functions that look like they should cause finalize to be used are problematic and at times buggy. Its invokation is not guaranteed!

Inheritance

- Since one of our primary goals with Java is to perform OOP
  - We always create inheritance hierarchies!
  - In fact, every class, unless otherwise requested, is derived from Java's standard root class Object
  - To derive a class from a base class in Java means that you are "extending" it

```
class list { //members}
public class ordered_list extends list {
        //more members – replacing old, adding new }
```

Accessing Base Class Members

- If a derived class has the same named member as the base class
  - It can be accessed by using the super keyword.
  - If we have a “cleanup” type function to be executed at the end of an object’s lifetime, it would need to use the super keyword to invoke it’s base class’s (I recommend that you first cleanup your derived class prior to invoking the base class’ cleanup

```
public class ordered_list extends list {
      public void member() {
         super.member();  //calls base class member
      }
}
```

Is there Hiding? Yes and No

- Hiding exists like it does for C++ for data members (fields)
- But, a derived class member function with the same name as a base class member function will not hide the base class' member!
- This means that funciton overloading in Java works between classes in a hierarchy
  - Which is what we "wished" happened in C++!

Constructors in Hierarchies

- Default constructors for base classes are implicitly invoked from the derived class' constructor
  - As with C++, from the base class "outward"
- However, when we have constructors with arguments, this gets more complex (but of course is handled differently than C++!)
  - In Java, we must explicitly write the calls to the base class constructor using the super keyword, followed by the appropriate arguments:
  - This must be the first thing that is done in your derived class constructor
  - Luckily, Java will complain if you don't do this! Unlike C++.

```
public class ordered_list extends list {
        ordered_list(int i) {
                super(i);               //causes base class constructor with an int to be called
```

If you have a finalize()...

- Within a hierarchy, if you need finalize() in a derived class and base class
  - It is important to remember to call the base class' version of finalize()
  - Otherwise, the base class finalization will not happen!

```
//In the derived class
protected void finalize() {
    super.finalize();
    }
```

A Java Application

- Must contain a method called `main()`
- Execution always begins with the first statement in method `main()`
- Any other methods in your program are subprograms and are not executed until they are sent a message

Where do we get started? main

- Unlike C++,
  - stand alone programs must have at least one class
  - it must have the same name as the file and
  - within that class must be a method called main!

  public static void main(String[] args)
- The public keyword means that the member function (method) is available to the outside world
- The static keyword means that this is a static member function which does not need a object of its class inorder to be invoked
- The argument is required (whether or not it is used) which holds the command line arguments.
- In C++ the command line arguments are optional as part of main
  - int main (int argc, char * argv[]);
  - Where argv is a “ragged array” in C and C++ (an array of arrays of characters)
  - In Java args is an array of string object references

Main in which class?

- Now that we have a hierarchy, where does main go?
- Well, you can put it in each class so that you can independently test
- The appropriate main is invoked based on which class name is used on the command line

I

Final Methods – a special case

- Final Methods?
  - Means that any inheriting class cannot change its meaning
  - It means that the method cannot be overridden
  - Allows for any calls to this method to be inline for better efficiency
  - Turns off dynamic binding
  - All private members are implicitly “final”
    - Because if you can’t access a private method so you couldn’t override it!

    public final void func() { //body of the function}

CS202 Introduction to Java

- Final Classes?
  - Means that no classes can be derived from this class (or inherit from this class)
  - For security reasons you do not want any subclassing…
  - Or, you want to make it as efficient as possible
  - Therefore, all methods are implicitly final

More Definitions

- Override When an instance method in a derived class has the same form of heading as an instance method in its superclass, the method in the derived class overrides (redefines) the method in the superclass
- Hide When a field in a derived class has the same name as one in its superclass or a class method has the same form of heading as a class method in its superclass, the field or class hide the corresponding component in the superclass

  Say again?

An example

```
public class Example
{
  char letter;
  public static String lineIs();
…
}
public class ExtExample extends Example
{
  char letter;
  public static String lineIs();
…
}
```

Hiding or overriding?

Another Example

```
public class Example
{
  char letter;
  public String lineIs();
…
}
public class ExtExample extends Example
{
  String letter;
  public String lineIs();
…
}
```

Hiding or overriding?

Class Syntax

Derived Class Syntax

```
ClassModifier class Identifier extends ClassName
{
    ClassDeclaration
    . . .
}
```

Overriding vs. Hiding

- We ***override*** an instance method of a superclass by providing an instance method in a derived class with the same form of heading

- We ***hide*** a data field of a superclass by providing a field in a derived class with the same name

Polymorphism

- Polymorphism is the ability of a language to have duplicate method names in an inheritance hierarchy and to decide which method is appropriate to call depending on the class of the object to which the method is applied.

Dynamic Binding

- All methods are bound in Java using run-time dynamic binding
  - Unless the method (or class) is “final”
  - So, we can use upcasting as we did in C++ to produce desired dynamic binding effects:

    list obj = new ordered_list();
    - Here, an ordered list object is created and the reference is assigned to a list reference

    obj.display(); //won’t call List’s display but rather ordered_lists!

Dynamic Binding

- Java, like C++, has some rules to get dynamic binding to work for us
  - The methods must be defined in the base class (to which we use a reference to) and they must be anything BUT private (public, protected, or “friendly” are all ok)
  - We must invoke the function thru a reference to the base class, but have it refer to an object of the proper class to which we are interested
  - The argument lists, function names, and return types must be identical
  - The only difference is we don’t need the “virtual” keyword (that was C++)

Overriding or Overloading?

- When you derive from a base class and implement a method that is in the base class
  - If the arguments and return type are the same you are overriding it
  - If the arguments are different, you are overloading!
  - This is very hard to debug since no other mechanism establishes dynamic binding

Abstract Base Classes

- Abstract base classes can help with this issue
  - Because if the methods from them are ever directly called you will find out immediately that something is wrong
  - The intent, as with C++, is to create a common interface
  - So that the derived classes can express their uniqueness!
  - All derived class methods that match the signature of the base class will be called using dynamic binding
  - This is created by making one or more abstract methods in the base class:

```
abstract void func();          //with no body
```

Abstract Base Classes

- If a class has just one of these abstract methods, the class must be qualified as “abstract”, otherwise you get an error:

```
abstract class list{
    public abstract void display();
}
```

- To inherit from an abstract class (and you want objects to exist of your class),
  - you must implement all of the functions that are abstract in the base class
- An abstract class without any abstract methods
  - means that you just can’t create any objects of that class!

Interfaces in Java

- The interface keyword creates a completely abstract class
  - One that provides for no implementation
  - Makes it “pure”
  - It allows us to specify the method names, argumetn lists, and return types – but no bodies
  - It can include data members, but they are always implicitly static and final
  - It provides a “form” rather than an implemented class
  - Use the “interface” keyword instead of the class keyword
  - All of the members are automatically “public” even if you don’t use the keyword. They are never “friendly” and cannot be protected or private!

“Implements” in Java

- The implements keyword allows classes to “derive” from a completely abstract class or to “implement” the code for a pure abstract class
- The implementation becomes an ordinary class which can be extended in the regular way
- Except that members must all be defined as public

```
interface list { void display(); }
class ordered_list implements list {
    Public void display();}
```

Multiple Inheritance!

- Since an interface has no memory and has not implementation,
  - There is nothing that prevents us from having classes implement more than one interface!
  - If you inherit from a non-interface, you can inherit from only one
  - The “extended” class comes first and the implementation of interfaces must come second, in a class doing both:

```
class ordered_list extends list
    implements one, two, three { ….}
```

What to use? So many choices!

- Even if you are not using multiple inheritance
  - Interfaces are preferable to abstract classes which in turn are preferable to concrete classes when thinking about a common base class
  - As we discussed in C++, if you are doing dynamic binding, it is best if all methods are dynamically bound – otherwise you will get stuck with having to know the data type you are dealing with at run time (RTTI)
  - An interface ensures that this is the case

Shadowing

- Shadowing A scope rule specifying that a local identifier declaration blocks access to an identifier declared with the same name outside the block containing the local declaration

- A shadowed class member can be accessed by using keyword `this` together with the class member

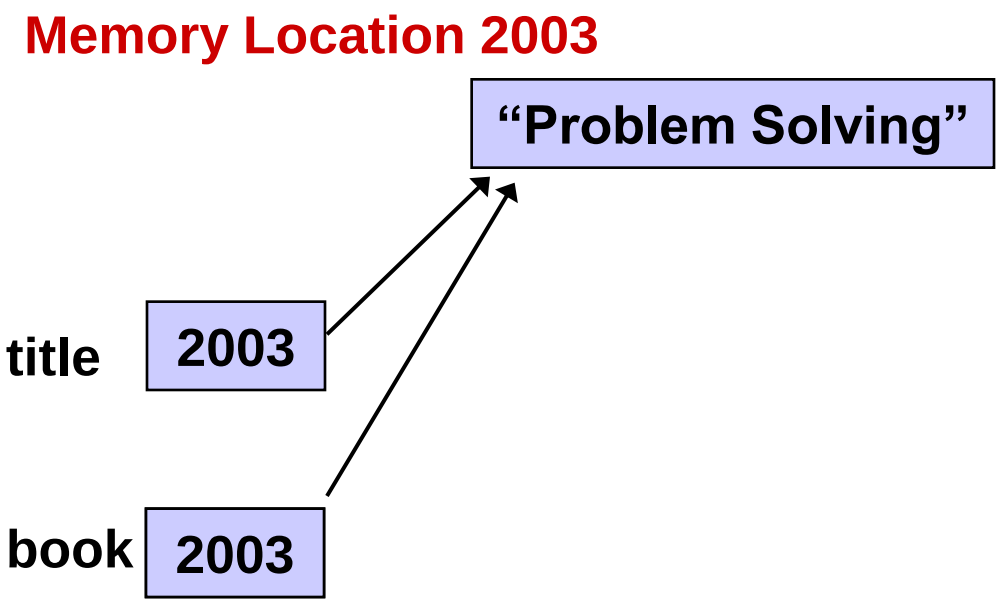
Input and Output

- To perform I/O in Java requires invoking a method as part of the System class
  - out is a static PrintStream object
  - Because it is static, you do not need to reference it through an object of class System (but can reference it via the class name instead)
  - The println method displays the information followed by a newline
  - System.out.println(“stuff”);

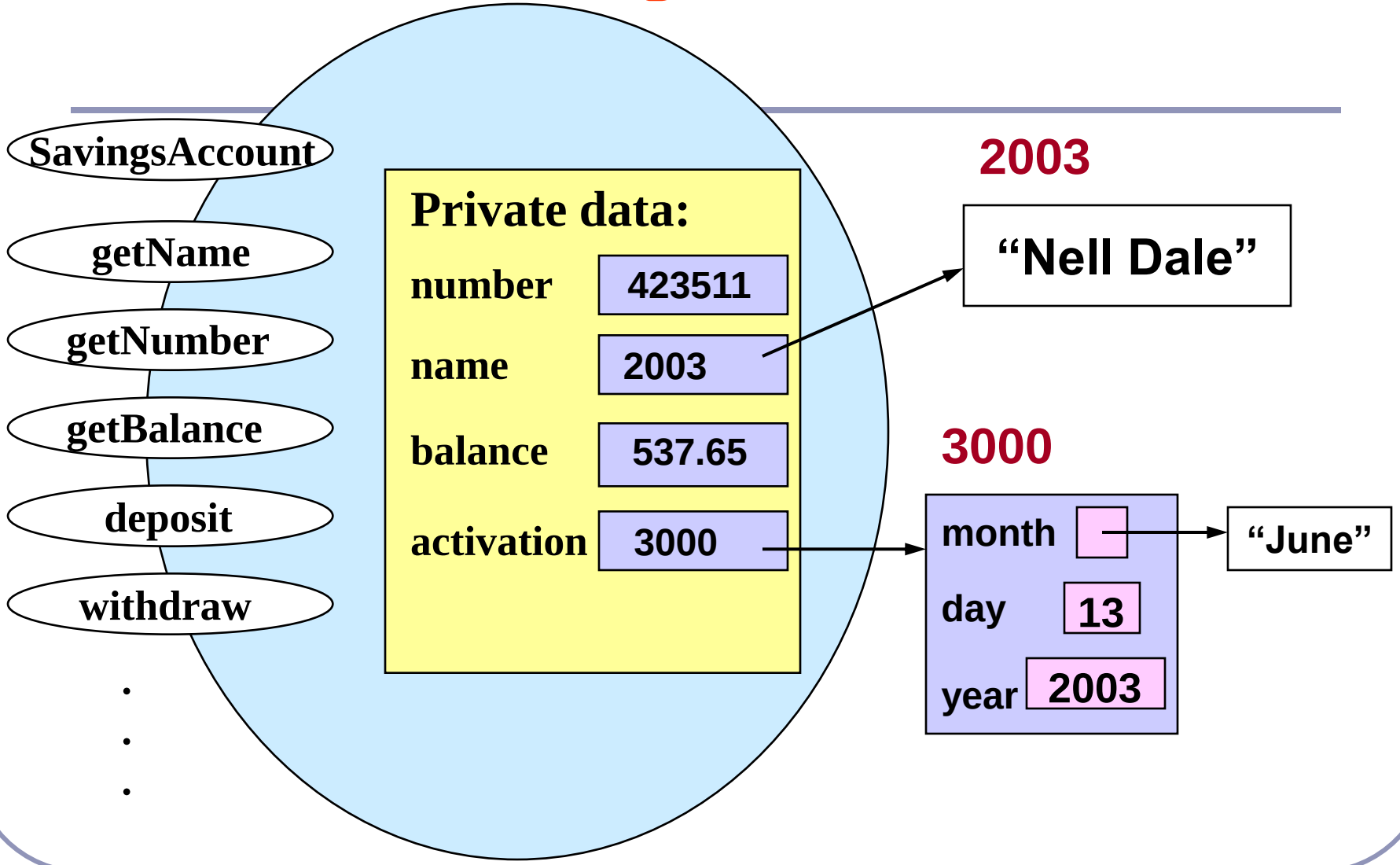
Reference Types: A Review

- A variable of reference type stores the address of the location where the object can be found.

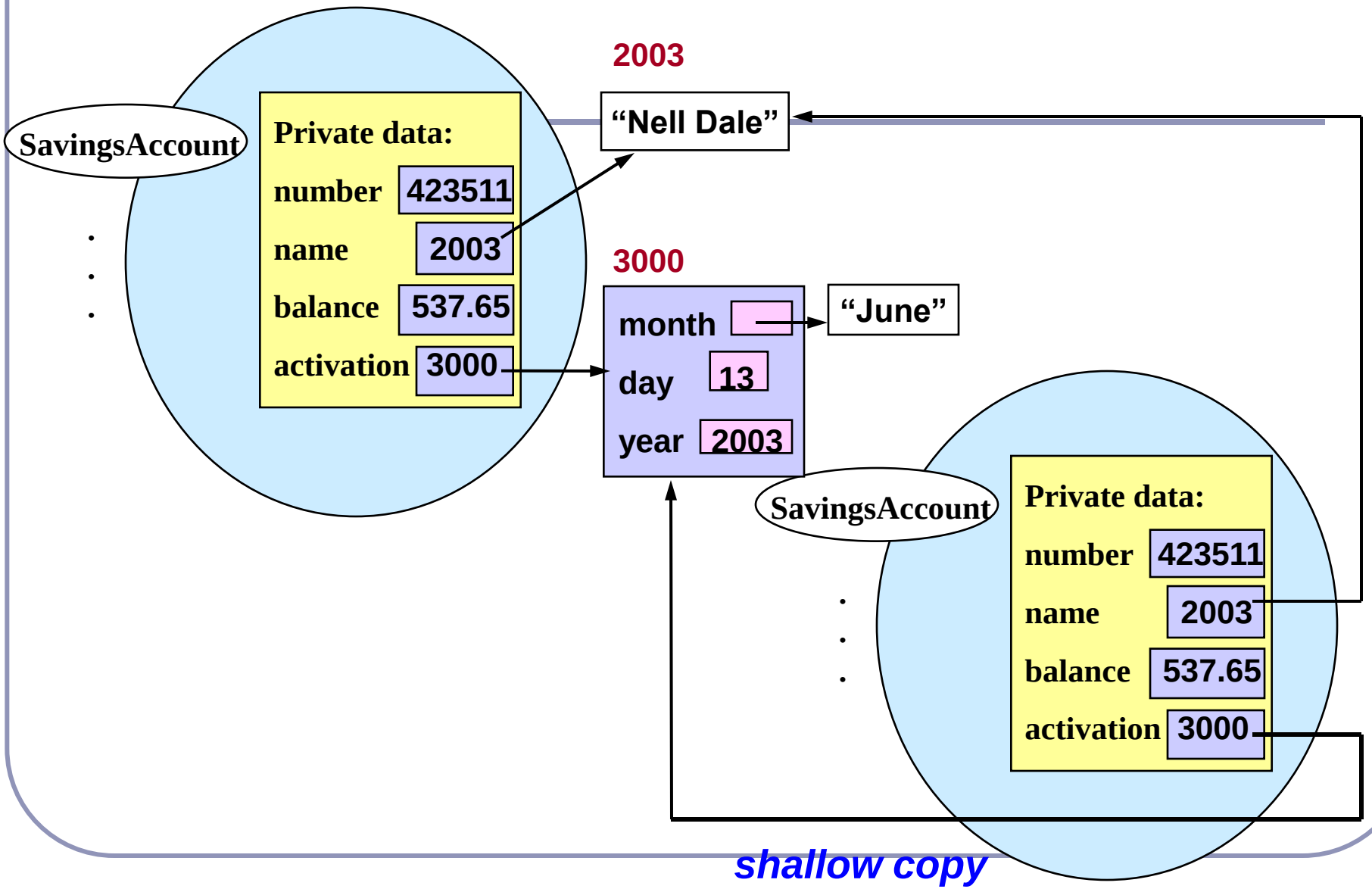
```
String  title ;

String  book ;

title = “Problem Solving”;

book = title ;
```



Class SavingsAccount



Shallow Copy



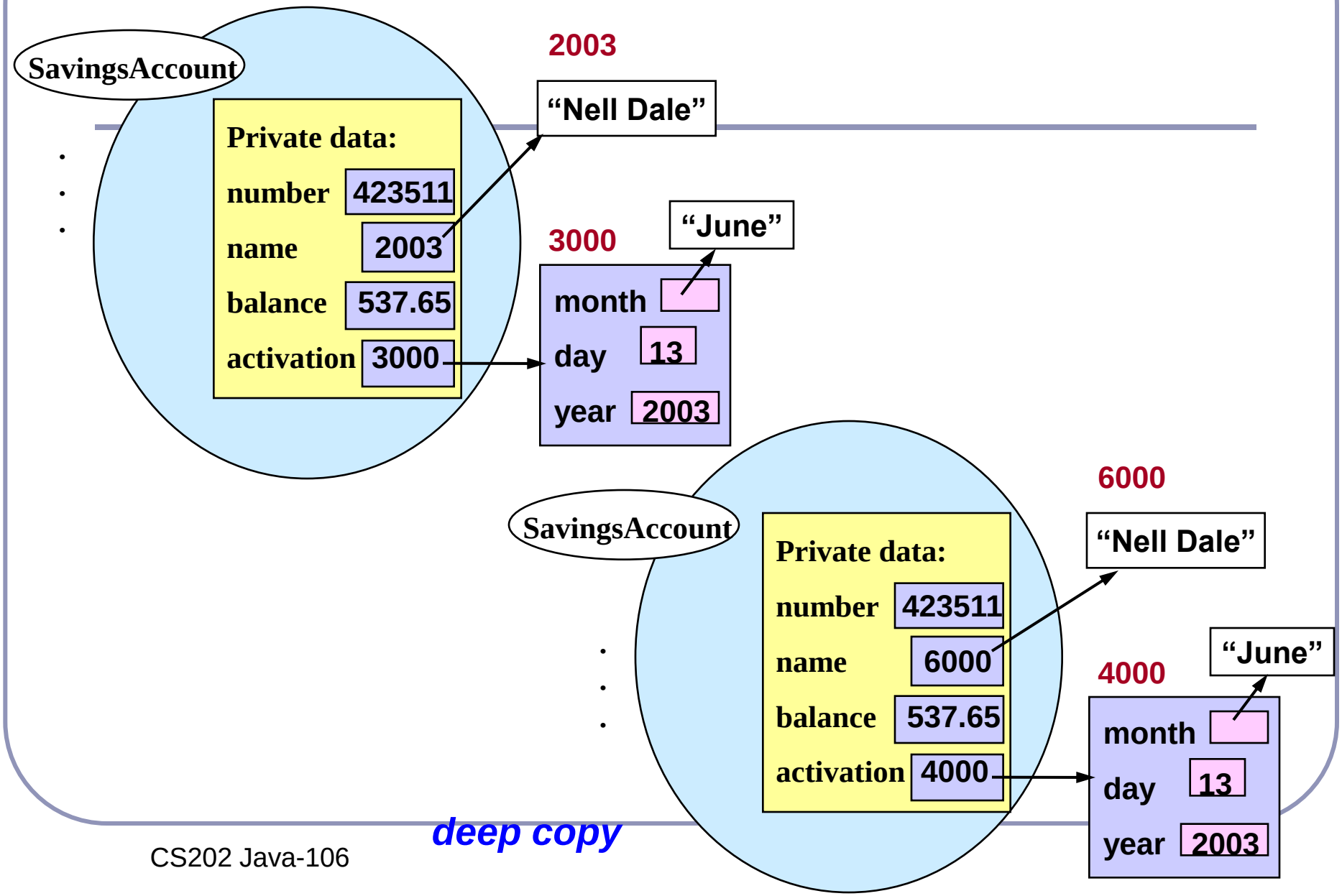
Shallow Copy vs. Deep Copy

- Shallow copy All class data fields, including references are copied; any objects referred to by data fields are not copied
- Deep copy All class data fields are copied, and all objects referred to are copied

What's the difference?

- A shallow copy shares nested objects with the original class object
- A deep copy makes its own copy of nested objects at different locations than in the original class object

Separate deep copy



Copy Constructor: Different...

- A copy constructor is a constructor that creates a deep copy of an object that can be used for other purposes, such as creating a new instance of an immutable object from an old one

```
public SavingsAccount(SavingsAccount oldAcct,
                      String changeOfAddress)
{
    . . .              // create deep copy of oldAcct
}
```

```
// call
account = new Savings Account(oldAcct, newAddress);
```

Java `String` Class

- A **string** is a sequence of characters enclosed in **double quotes.**
- **string sample values**

  `"Today and tomorrow"`

  `"His age is 23."`

  `"A"` (a one character string)
- The empty string contains no characters and is written as `""`

Actions of Java's `String` class

- **String operations include**
  - **joining one string to another (concatenation)**
  - **converting number values to strings**
  - **converting strings to number values**
  - **comparing 2 strings**

- ***Why is `String` uppercase and `char` lower case?***
  - **`char` is a built in type**
  - **`String` is a class that is provided**
  - **Class names begin with uppercase by convention**

Assignment Statement Syntax

```
Variable = Expression;
```

First, Expression on right is evaluated.

Then the resulting value is stored in the memory location of Variable on left.

NOTE: The value assigned to Variable must be of the same type as Variable.

String concatenation (+)

- **Concatenation uses the + operator.**
- **A built-in type value can be concatenated with a string because Java automatically converts the built-in type value for you to a string first.**

Concatenation Example

```
final  int  DATE = 2003;
final  String  phrase1 = "Programming and Problem ";
final  String  phrase2 = "Solving in Java ";
String bookTitle;

bookTitle = phrase1 + phrase2;
System.out.println(bookTitle + " has copyright " + DATE);
```

Using Java output device

METHOD CALL SYNTAX

System.out.print (*StringValue*);

System.out.println (*StringValue*);

These examples yield the same output.

```
System.out.print("The answer is, ");
System.out.println("Yes and No.");

System.out.println("The answer is, Yes and No.");
```

Java Input Devices

- More complex than Output Devices
- Must set one up from a more primitive device

```
InputStreamReader inStream;
inStream = new InputStreamReader(System.in);
// declare device inData
BufferedReader inData;
inData = new BufferedReader(inStream)
```

Using a Java Input Device

```
// Get device in one statement
inData = new BuffredReader(new
  InputStreamReader(System.in));
String oneLine;
// Store one line of text into oneLine
oneLine = inData.readLine();
```

Where does the text come from?

Interactive Input

- `readLine` is a value-returning method in class `BufferedReader`
- `readLine` goes to the `System.in` window and inputs what the user types
- How does the user know what to type?
- The program (you) tell the user using `System.out`

Interactive Output continued

```
BufferedReader inData;
inData = new BufferedReader(new
  InputStreamReader(System.in));
String name;
System.out.print("Enter name: ");
name = inData.readLine();
```

Name contains what the user typed in response to the prompt

Inputting Numeric Values

- ***If `readLine` inputs strings, how can we input numbers?***
- **We convert the strings to the numbers they represent.**

  "69.34" becomes 69.34

  "12354" becomes 12354
- ***Yes, but how?***

Predefined Numeric Classes

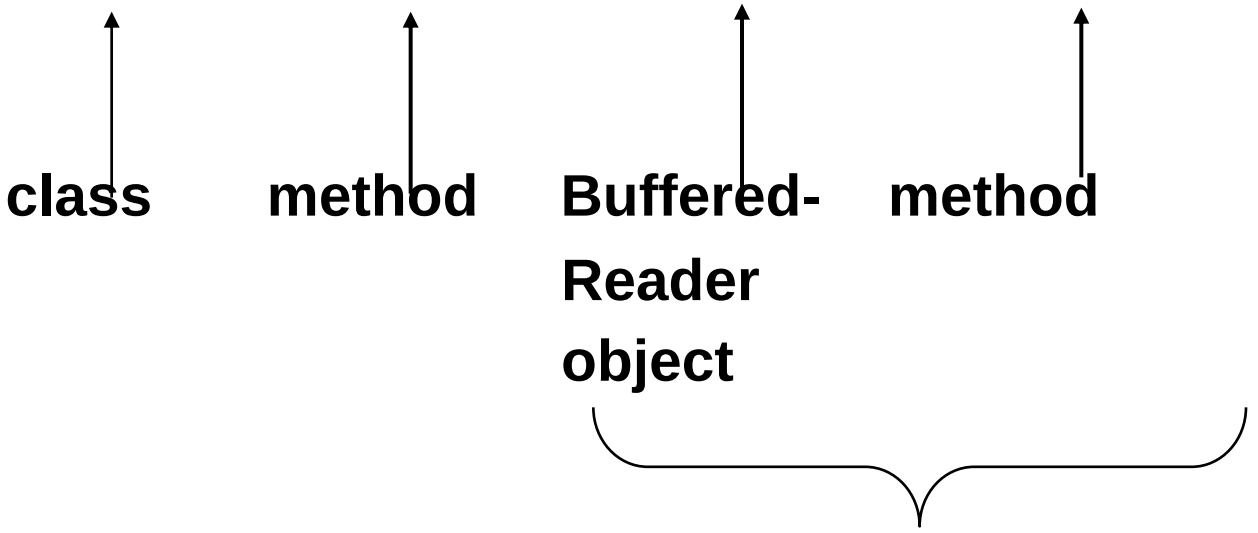
Built-in Type	Class
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

`parseInt, parseLong, parseFloat, parseDouble`

are class methods for translating strings to numeric values

Converting Strings to Numbers

```
int intNumber;
System.out.println("Enter an integer: ");
intNumber =
  Integer.parseInt(inData.readLine());
```



Converting a String to a Double Value

```
double  price ;
```

String object

```
price = Double.parseDouble(inData.readLine());
```

string converted to double value

Java Program

```
// ********************************************************
//  PrintName prints a name in two different formats
// ********************************************************
public class PrintName
{
  public static void main (String[ ] args)
  {
    BufferedReader inData;
    String first;              // Person's first name
    String last;               // Person's last name
    String firstLast;          // Name in first-last format
    String lastFirst;          // Name in last-first format
    inData = new BufferedReader(new
        InputStreamReader(System.in));
```

Java program continued

```
        System.out.print("Enter first name: ");
        first = inData.readLine();

        System.out.print("Enter last name: ");
        last = inData.readLine();

        firstLast = first + " " + last;
        System.out.println("Name in first-last format is "
                           + firstLast);
        lastFirst = last + ", " + first;
        System.out.println("Name in last-first format is "
                           + lastFirst);

    }
}
```

Additional String Methods

- Method `length` returns an `int` value that is the number of characters in the string

```
String name = "Donald Duck";
numChars;
numChars = name.length();
```

instance method

length is an instance method

String Methods Continued

- Method `indexOf` searches a string to find a particular substring, and returns an `int` value that is the beginning position for the first occurrence of that substring within the string
- Character positions begin at 0 (not 1)
- The substring argument can be a literal `String`, a `String` expression, or a `char` value
- If the substring could not be not found, method `indexOf` returns value `-1`

String Methods Continued

- Method `substring` returns a substring of a string, but does not change the string itsel
- The first parameter is an `int` that specifies a starting position within the string
- The second parameter is an `int` that is 1 more than the ending position of the substring
- Remember: positions of characters within a string are numbered starting from 0, not from 1.

What value is returned?

```
// Using methods length, indexOf, substring
     String  stateName = "Mississippi";

     stateName.length();                    ?

     stateName.indexOf("is");               ?

     stateName.substring(0, 4);             ?

     stateName.substring(4, 6);             ?

     stateName.substring(9, 11);            ?
```

Relational operators w/Strings?

Remember that strings are reference types

```
myString = "Today";
yourString = "Today";

myString == yourSring
```

returns what?

String methods

Method Name	Parameter Type	Returns	Operation Performed
equals	String	boolean	Tests for equality of string contents.
compareTo	String	int	Returns 0 if equal, a positive integer if the string in the parameter comes before the string associated with the method and a negative integer if the parameter comes after it.

```
String   myState;
String   yourState;

myState = "Texas";
yourState = "Maryland";
```

EXPRESSION	VALUE
myState.equals(yourState)	false
0<myState.compareTo(yourState)	true
myState.equals("Texas")	true
0>myState.compareTo("texas")	true

More String Methods

Method Name	Parameter Type	Returns	Operation Performed
toLowerCase	none	String	Returns a new identical string, except the characters are all lowercase.
toUpperCase	none	String	Returns a new identical string, except the characters are all uppercase.

String Method `compareTo`

- When comparing objects with ==, the result is true only if both references refer to the same object in memory
- String method **compareTo** uses a dictionary type comparison of strings and **returns**
  - 0 if they have the same letters in the same order
  - a negative number if the instance string is less than the string passed as a parameter
  - a positive number if the instance string is greater than the string that is passed

Values of each expression

```
String s1 = new String("today");
String s2 = new String("yesterday");
String s3 = new String("today");
String s4 = new String("Today");
```

`s1.compareTo(s2)`	`-5`
`s1 == s3`	`false`
`s1.compareTo(s3)`	`0`
`s1.compareTo(s4)`	`32`

Example of If statements (same use of {}, if and else)

```
if (creditsEarned >= 90)
  System.out.println("Senior Status");

else if (creditsEarned >= 60)
  System.out.println("Junior Status");

else if (creditsEarned >= 30)
  System.out.println("Sophomore Status");

else
  System.out.println("Freshman Status");
```

A sentinel-controlled loop

- Requires a “priming read”
- “Priming read” means you read one data value (or set of data values) before entering the *while* loop
- Process data value(s) and then read next value(s) at end of loop

```
// Sentinel is negative blood pressure.
int thisBP; int total; int count;
count = 1;                  // Initialize
total = 0;
// Priming read
thisBP = Integer.parseInt(dataFile.readLine());
while (thisBP > 0)          // Test expression
{
  total = total + thisBP;
  count++;                  // Update
  thisBP = Integer.parseInt(dataFile.readLine());
}

System.out.println("The total = "  +  total);
```

An end-of-file controlled loop

- **depends on fact that `readLine` returns `null` if there is no more data**

```
// Read and sum until end of line
int thisBP; int total; int count;
count = 1;                  // Initialize
total = 0; String line;
line = dataFile.readLine();
while (line != null)        // Test expression
{
  thisBP = Integer.parseInt(line);
  total = total + thisBP;
  count++;                  // Update
  line = dataFile.readLine();
}
System.out.println("The total = "  +  total);
```

Flag-controlled loops

- Use meaningful name for the flag
- Initialize flag (to true or false)
- Test the flag in the loop test expression
- Change the value of the flag in loop body when the appropriate condition occurs

A flag-controlled loop

- Count and sum the first 10 odd numbers in a data file
- Initialize flag `notDone` to `true`
- Use `while(notDone)` for loop test
- Change flag to `false` when 10 odd numbers have been read or if EOF is reached first

```
count = 0;
sum = 0;
notDone = true;
while ( notDone )
{
  line = dataFile.readLine( );   // Get a line
  if (line != null)              // Got a line?
  {
    number = Integer.parseInt(line);
    if (number % 2 == 1)         // Is number odd?
    {
      count++;
      sum = sum + number;
      notDone = ( count < 10 );
    }
  }
  else                           // Reached EOF unexpectedly
  {
    errorFile.println("EOF reached before ten odd values.")
    notDone = false;             // Change flag value
  }
}
```

Exception Handling

- Handling errors is always a difficult problem
  - Many times we ignore error handling
  - Think back to your 162 project where this was important? What did it do to your design?
  - A major problem with most error handling schemes is that they rely on the programmer's vigilance and an agreed upon convention ahead of time – which may not be enforced by the language
- Exception handling in Java is handled directly as part of the language
  - And, you are forced to use it to get anywhere
  - If you don't write your code to properly handle exceptions, you will get error messages!
  - This consistency makes error handling easier
- Side note: Error handling is not an object oreiented feature!

Exceptions

- An exception is an unusual situation that occurs when the program is running.
- Exception Management
  - **Define the error condition**
  - **Enclose code containing possible error (try).**
  - **Alert the system if error occurs (throw).**
  - **Handle error if it is thrown (catch).**

Three Part Exception Handling

- Defining the exception

  Extend type Exception and supply a pair of constructors that call super

- Raising(generating) the exception

  Use of the throw statement

- Handling the exception

  Forward the exception or use try-catch-finally statement to catch and handle the exception.

try-catch with Built-In Exception

```
filename = fileField.getText();

try
{
  outFile = new PrintWeriter(new FileWriter(filename));
}
catch(IOException except)
{
  errorLabel.setText("Unable to open file " + filename);
  fileField.setText("");
}
```

try-catch with Exception Class

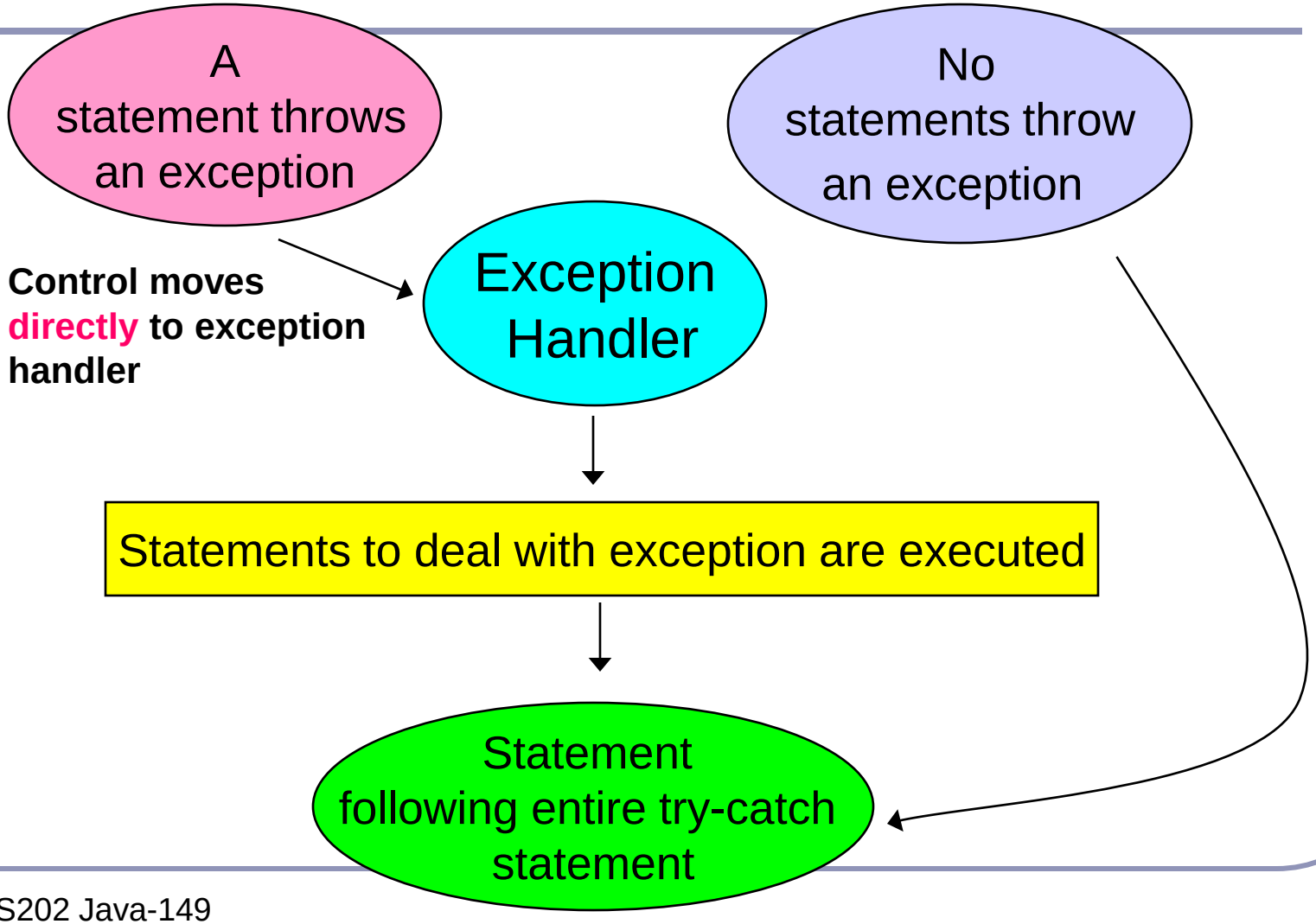
```
try
{
   . . .    // Statements that might contain an error
  throw new DataException("bad data");
}
catch(DataException except)
{
  System.out.println(except.getMessage());
}
```

What is Class DataException?

Class DataException

```
public class DataException extends Exception
{
  public DataException()
  {
    super();
  }
  public DataException(String message)
  {
    super(message);
  }
}
```

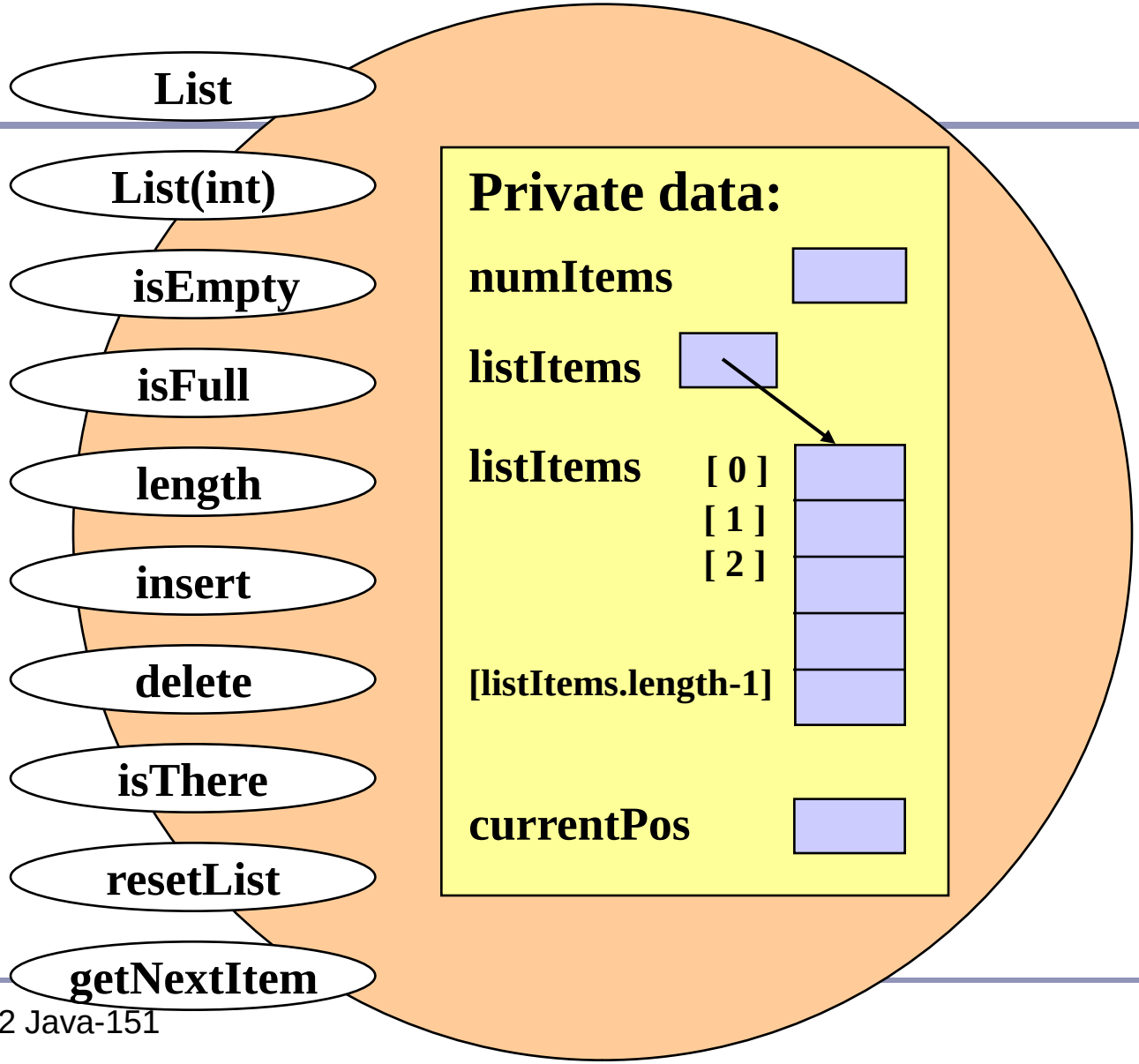
Execution of *try-catch*



Precedence

Operator	*Associativity*
()	Left to right
unary: ++ –– ! + – (cast)	Right to left
* / %	Left to right
+ –	Left to right
< <= > >=	Left to right
== != &	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
? :	Right to left
= += –= *= /=	Right to left

class List


List
List(int)
isEmpty
isFull
length
insert
delete
isThere
resetList
getNextItem
Private data:
numItems
listItems
listItems
[0]
[1]
[2]
[listItems.length-1]
currentPos

class List

```
// class List
public class List
{
   // Data fields
   protected String[] listItems;
   // Array to hold list items

   protected int numItems;

   // Number of items currently in list

   protected int currentPos;

   // State variable for iteration
        .   .   .
}
```

Unsorted and Sorted Lists

UNSORTED LIST	SORTED LIST
Elements are placed into the list in no particular order with respect to their content	List elements are in an order that is sorted by the content of their keys -- either numerically or alphabetically

Methods for Class List

```
public List()      // Default Constructor
// Result: List instantiated for 100 items
{ numItems = 0;
  listItems = new String[100];
  currentPos = 0;
}
public List(int maxItems) // Constructor
// Result: List instantiated for maxItems items
{ numItems = 0;
  listItems = new String[maxItems];
  currentPos = 0;
}
```

Observer Methods

```
public boolean isEmpty()
// Returns true if no components; false otherwise
{
  return (numItems == 0)
}
public int length()
// Returns the number of components in the list
{
  return numItems;
}
```

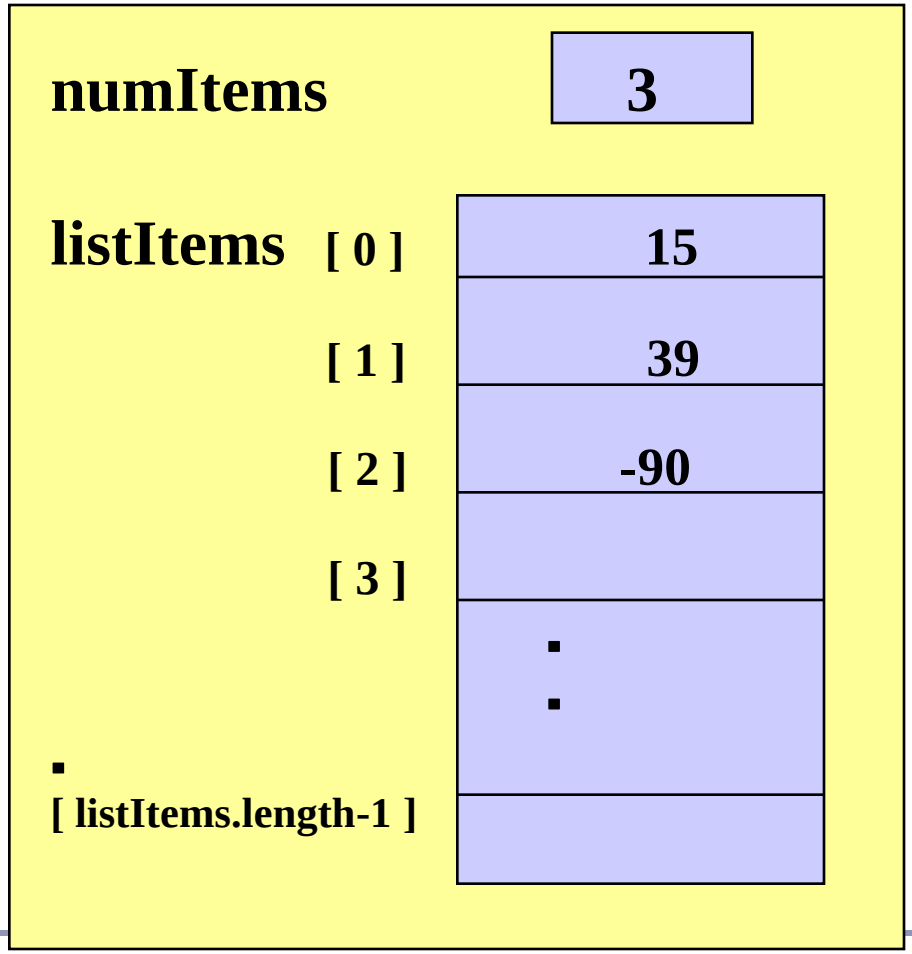
Observer Methods Contd.

```
public boolean isFull()
// Returns true if no more room; false otherwise
{
  return (numItems == listItems.length);
}
```

Transformer Method Insert

```
public void insert(String item)
// Result: If the list is not full, puts item in
// the last position in the list; otherwise list
// is unchanged.
{
   if (!isFull())
   {
      listItems[numItems] = item;
      numItems++;
   }
}
```

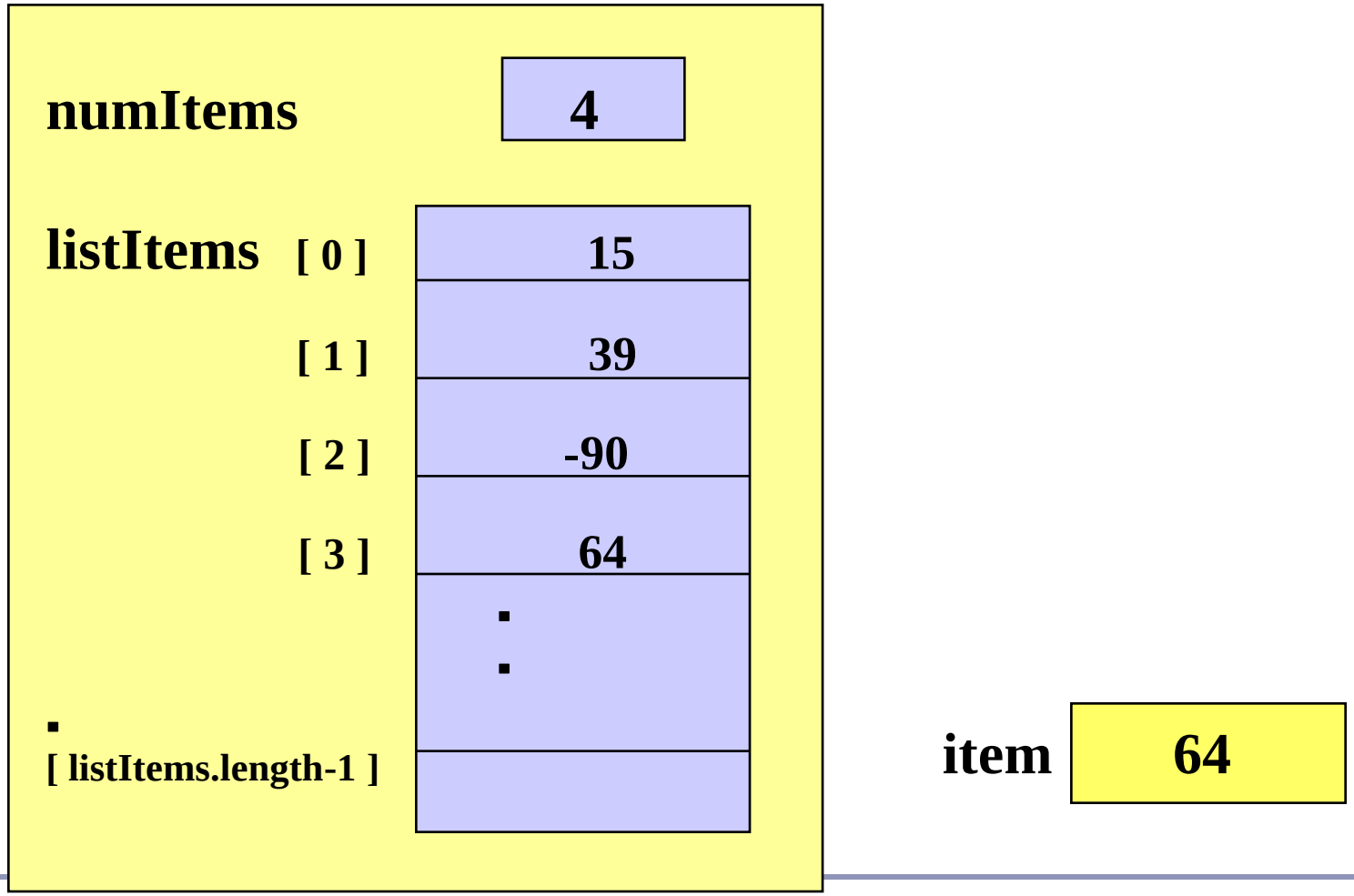
Before Inserting 64 into an Unsorted List



The item will be placed into the numItems location, and numItems will be incremented.

item 64

After Inserting 64 into an Unsorted List



Observer Method `isThere`

```
public boolean isThere(String item)
// Returns true if item is in the list; false otherwise

{
  int index  =  0;
  while (index < numItems  &&
      listItems[index].compareTo(item) != 0)
    index++;
  return (index < numItems);
}
```

Transformer Method Delete

- Find the position of the element to be deleted from the list
- Eliminate the item being deleted by shifting up all the following list elements
- Decrement `numItems`

```
public void delete(String item)
// Result: Removes item from the list if it is
// there; otherwise list is unchanged.
{
  int index = 0;
  boolean found = false;
  while (index < numItems && !found)
  {
    if (listItems[index].compareTo(item) == 0)
      found = true;
    else
      index++;
  }
```

```
// If item found, shift remainder of list up
   if (found)
   {
     for (int count = index; count < numItems - 1;
         count++)
       listItems[count] = listItems[count + 1];
     numItems--;
   }
}
```

Iterator Methods

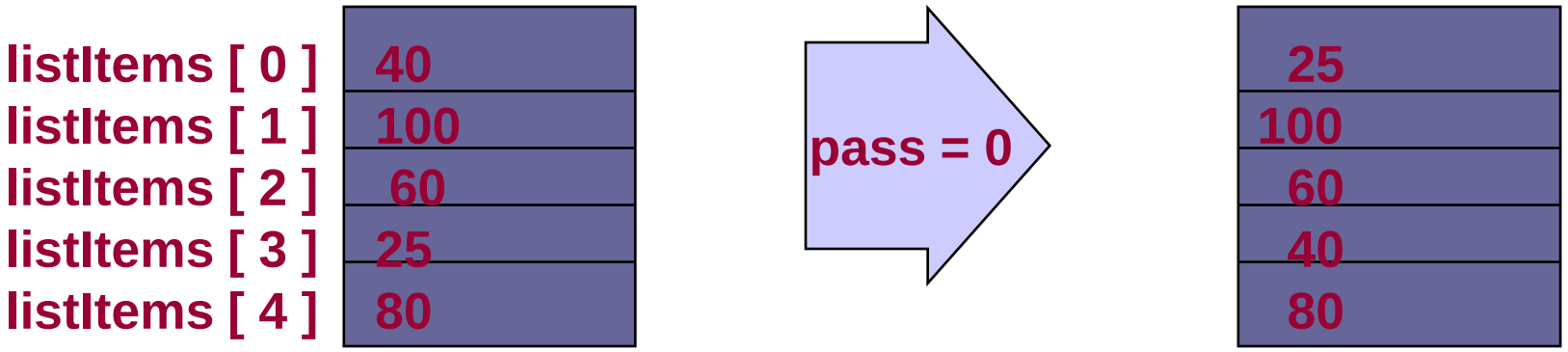
```
public void resetList()
// Initialize iterator by setting currentPos to 0
{
   currentPos = 0;
}
public String getNextItem()
// Returns current item; increments currentPos circularly
// Assumption: No transformers invoked since last call
{
  String next = listItems[currentPos];
  if (currentPos == numItems - 1)
    currentPos = 0;
  else
    currentPos++;
  return next;
}
```

Straight Selection Sort

- Examines the entire list to select the smallest element; places that element where it belongs (with array index 0)
- Examines the remaining list to select the smallest element from it; places that element where it belongs (array index 1)
- Continues process until only 2 items remain in unsorted portion
- Examines the last 2 remaining list elements to select the smallest one; place that element where it belongs in the array (index `numItems-2`, leaving the last item in its proper place as well.

Selection Sort Algorithm

FOR passCount going from 0 through numItems - 2
Find minimum value in listItems [passCount] . .
listItems[numItems-1]
Swap minimum value with listItems [passCount]

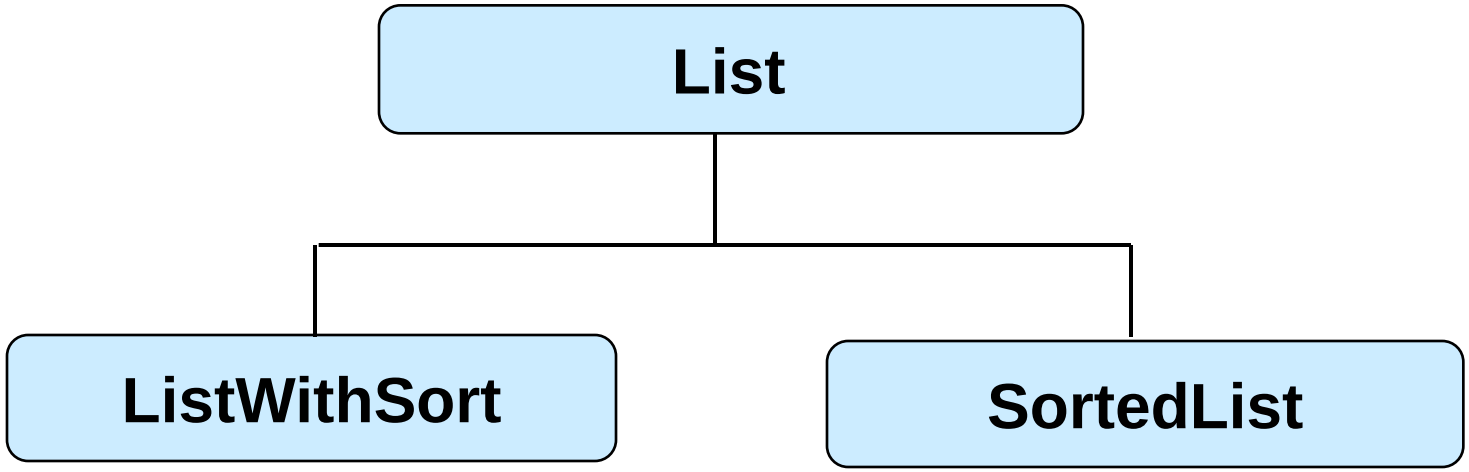


Selection Sort Code

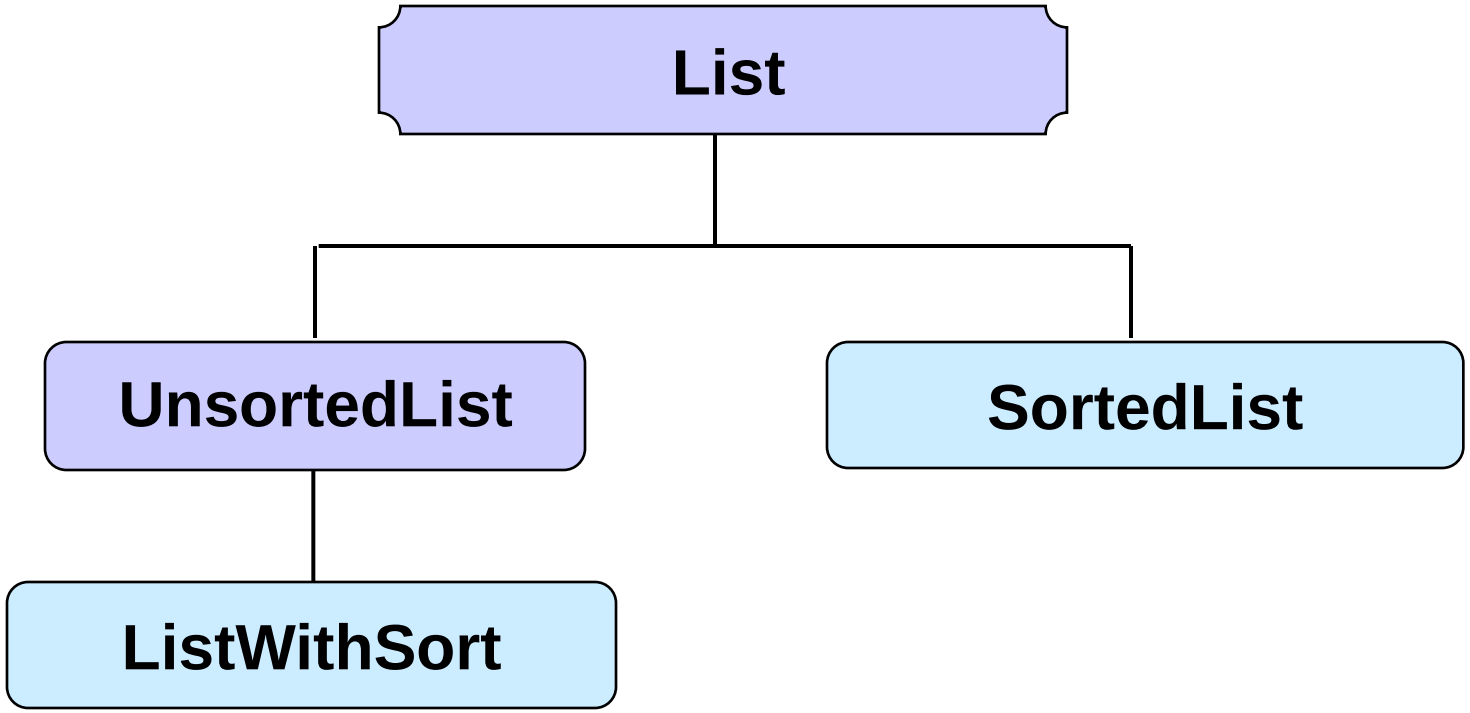
```
public void selectSort()
// Sorts array into ascending
{ String temp; int passCount; int sIndex;
  int minIndex;        // index of minimum so far
  for(passCount = 0; passCount < numItems-1; passCount++)
   { minIndex = passCount;
     //  find index of smallest remaining
     for(sIndex = passCount + 1; sIndex < numItems; sIndex++)
       if(listItems[sIndex].compareTo(listItems[minIndex])<0)
         minIndex = sIndex;
     temp = listItems[minIndex];          // swap
     listItems[minIndex] = listItems[passCount];
     listItems[passCount] = temp;
  }
}
```

```
public void insert(String item)
// If the list is not full, puts item in its proper
// place; otherwise list is unchanged.
// Assumption: item is not already in the list.
{ if (! isFull())
  {//   find proper location for new element
     int index = numItems - 1;
     while (index >= 0  &&
        item.compareTo(listItems[index]) < 0)
    {
       listItems[index + 1] = listItems[index];
       index--;
    }
    listItems[index +1] = item; // insert item
    numItems++; }
}
```

List class hierarchy



Abstract List Class Hierarchy



```
public boolean isThere(String item)
// Assumption: List items are in ascending order
// Returns true if item is in the list; false otherwise
{ int first = 0;  int last = numItems - 1;
  int middle;  boolean found = false;
   while (last >= first && !found)
   { middle = (first + last) / 2;
     if (item.compareTo(listItems[middle]) == 0)
        found = true;                  // Item  has been found
     else if (item.compareTo(listItems[middle] < 0)
        last = middle - 1;            // Look in first half
     else first = middle + 1;         // Look in second half
   }
    return found;
}
```

`Comparable` **Interface**

- **Is part of the standard Java class library**
- **Any class that implements the Comparable interface must implement method compareTo**
- **String implements the Comparable interface**

Let's build a stack class

```
public class Stack {
    private int[] data;
    private int ptr;

    public Stack(int size) {
        data = new int[size];
        ptr = 0;
    }
    public void push(int x) {
        if (ptr < data.length)
            data[ptr++] = x;
    }
```

```
public int pop() {
    if (ptr > 0)
        return data[--ptr];
    else
        return Integer.MIN_VALUE;
}
public int top() {
    if (ptr > 0)
        return data[ptr-1];
    else
        return Integer.MIN_VALUE;
}
public int size() {
    return ptr;
}
```

```
    public static void main(String[] args) {
        Stack stk = new Stack(5);
        for (int i = 0; i < 5; ++i)
            stk.push(i);
        while (stk.size() > 0)
            System.out.print(stk.pop() + " ");
        System.out.println();
    }
}

$ javac Stack.java
$ java Stack
4 3 2 1 0
$
```

Initializing Static Members

- This is very different from C++
- Remember, there is only one instance of a static data member no matter how many objects there are of the class
- We initialize them in a “static block”

```
class list {
        static int max_lists;
        static {
                max_lists = 100;
                }
```

A static block is executed only once: the first time you make an object of the class or access a static member.

Libraries

- To use a library we must import it:
  import java.util.*;
  - This brings in the entire utility library
  - And, it manages the namespace since it only brings in those classes that are requested
- When we have multiple source files, each file should have a public class that is the same name as the file (including capitalization but excluding the file extension)
- Multiple files can create a “package”
- If you place
  - package list_library;
  - As the first non-commented line in your file – you are saying that this compilation unit is part of a larger library that you are building
  - SO, there need not be a public class in each file since we are part of a larger unit.
  - And, then such packages can be imported!
  - If there are collisions in the libraries and packages that you import, then Java will require you to explicitly specify the “classpath” from which the class exists