String and Helper Classes

Objectives

- After this lecture, you will be able to:
  - understand stack and heap memory
  - declare and use String objects
  - declare and use StringBuilder
  - declare and use wrapper classes
  - declare and use high precision types

Review

What is the difference between .java file and a .class file?

a) A .class file contains Java code and a .java file contains byte code

b) A .class file contains byte code and a .java file contains Java code

c) A .class file contains one class and a .java file contains an entire program

d) Nothing; you can use either file extension

Review

What problem can data narrowing (downcasting) have?

a) Loss of precision

b) Increase in precision

c) Increase in errors

d) Nothing; downcasting is never a problem

Memory Types

- ***Stack*** memory: local primitive variables in methods
  - Created (pushed on the stack) when the method (including `main()`) is called, deleted when the method returns. This includes all local variables* and all primitive parameters
  - With the exception of static variables, memory cleanup is easy for the JVM: pop the stack
  - Memory is allocated at the top of the stack (push) and removed from the top of the stack (pop), so there are never any memory holes

**: object variables are on the stack, but not the object itself*

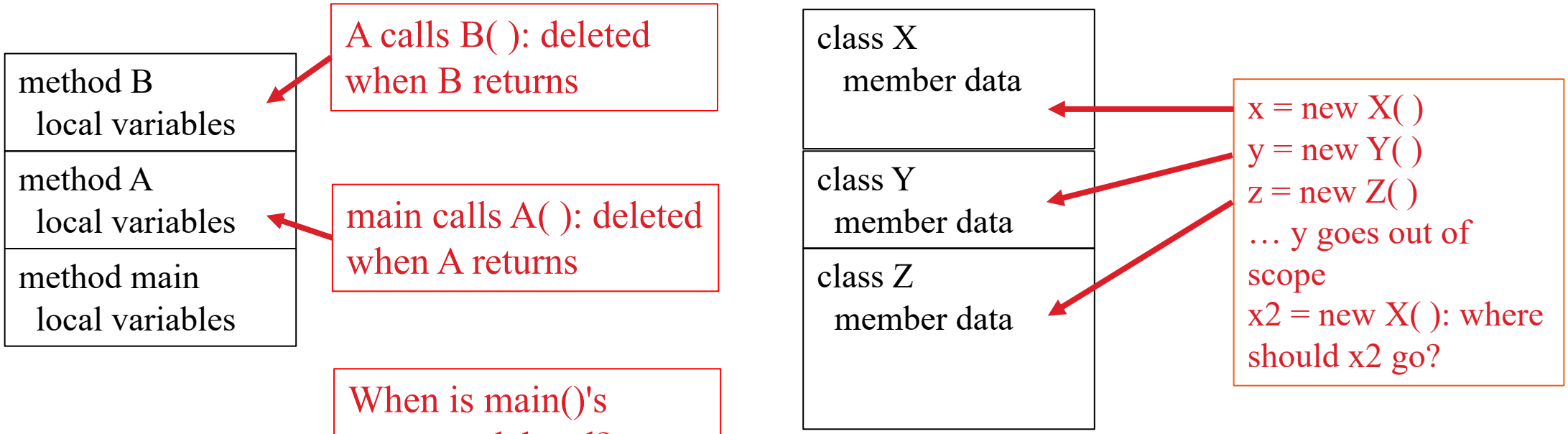
Memory Types, cont.

- ***Heap*** memory: for objects created from classes
  - Created when the ***new*** operator is invoked
  - ***new*** creates memory space to store the object's data (based on the class definition) and returns a reference to the location
  - Those references may last an indefinite amount of time
  - Technically, an object variable (on the stack) refers to (keeps the address of) the object (on the heap), after new-ing it
- Heap allocation creates a problem for the JVM: some memory is currently in use, some is not, and these sections can be interleaved

Stack versus Heap

Stack allocation:
method call and return

Heap allocation:
dynamic creation of objects



Garbage Collection

- The JVM has a ***garbage collector*** that, every once in a while, combs through the heap to reclaim and recycle unused memory: unreferenced objects
  - For long running programs that allocate and de-allocated heap memory, this is critical – the heap is finite, so you can run out of memory – and without a garbage collector, you could run out of memory even when there's freed-up memory
  - This is because it's too difficult to allocate freed memory that is located between in-use memory: you cannot guarantee that a contiguous chunk of memory big enough can be found

Garbage Collection, cont.

- So "reclaim and recycle" means ***defragmenting***: moving all the in-use memory up in the heap, moving all the freed memory back, and fixing all of the references to their new values
- This can take a lot of time (relatively speaking) for long-running programs
  - But not usually an issue for small student programs

Garbage Collection, cont.

- GC is a complicated thing
  - Don't try to influence Java's GC
  - In this course, don't worry about when the GC runs: it may never run out of memory, and so the GC will never be invoked
  - In production, this *might* be an issue, but don't fool with it unless you really, really know what you're doing

String Class

- The ***String*** class is a wrapper around char[ ] – that is, it contains an array of characters as member data, and provides a rich set of methods that access that data
- Some important methods are:
  - `length()`
  - `charAt(index)` – returns the char at position [index]
  - `indexOf(char), indexOf(String)` – returns the index of the char or (sub)String, or -1 if not found. Variants: `indexOf(char, startLookingHere), indexOf(String, startLookingHere)`
  - `replace(oldChar, newChar)` – returns a new String after replacement

String Class, cont.

String constructor called implicitly

String constructor called explicitly

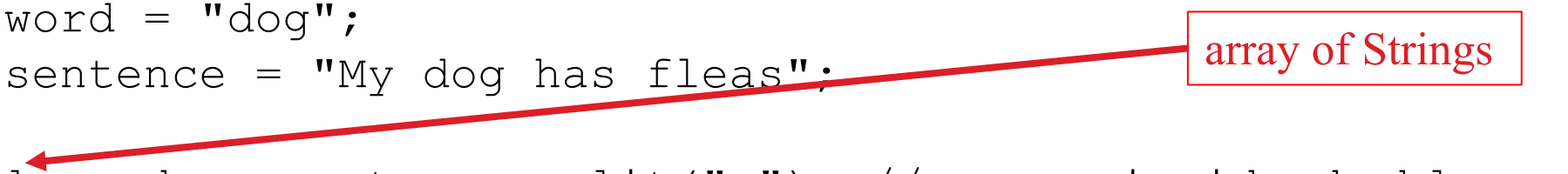
```
String word = "dog";
String sentence = new String();
sentence = scanner.next();              // Say the user enters"My
                                        //  dog has fleas";

System.out.println( word.charAt(1) );
System.out.println( word.indexOf( 'g' ) ); // Single quotes

System.out.println( sentence.indexOf( word ) );

String s = sentence.replace('a', 'X');
System.out.println( sentence );
System.out.println( s );
```

More String Methods

- `toUpperCase( ), toLowerCase( )` – returns a new String with all letters changed
- `trim( )` – removes front and back white space
- `equals(String), equalsIgnoreCase(String)` – returns true if the two strings' chars match, false otherwise – case-insensitive in the second version
  - ***Never*** use == to compare two Strings – it doesn't compare the char's
- `compareTo(String)` – returns a negative integer, 0, or positive integer if the base String is lexicographically less than, equal to, or greater than the parameter.
- `String[] split( regex )` – returns an array of String, split on the match

String Class, cont.

```
String word = "dog";
String sentence = "My dog has fleas";

String[] words = sentence.split(" "); // space inside double quotes
for (String str: words) {
    System.out.println(str);
}

String csv = "John,Doe,75000.00,Sales";
String[ ] employee = csv.split(","); // comma inside double quotes

for (String str: employee) {
    System.out.println(str);
}
```



String Class, cont.

- The String class is ***immutable*** – that is, its contents cannot be changed
  - Note, for example, that `replace( )` does not change the base String, it returns a new String
- String constants are a shortcut for declaring Strings, but remember that the String constructor is called
  - Java maintains these constants in a separate memory area of the heap, so that they can be reused, but there's not much gain in this

String Class, cont.

- Empty string: ""
- ***null*** string – never initialized or new'd
- An empty string can be operated on, but trying to do anything to a null string causes an exception
- Do this if you're not sure:

```
if (str != null) {
    str.<somemethod>( ); // or whatever
```

String Class, cont.

- String concatenation
  - + and += operators: concatenation and concat-and-assign
- Every time a new String is created, the String constructor is called
  - We'll talk more about constructors later
- For some applications, you might need a lot of strings, so this may slow things down

StringBuilder

- The `StringBuilder` class was invented for this situation
  - Create one `StringBuilder` object
  - Append String objects or character arrays (`char[ ]`) to it, often from an input object – like when reading a text file
  - The call `toString( )` to get one big String object back

StringBuilder example

```
StringBuilder sb = new StringBuilder();

... some code to open a text file ...

while (myfile.hasNextLine()) {
      sb.append( myfile.nextLine() + "\n";
}

String allOfIt = sb.toString( );       // One big String
String[] lines = allOfIt.split("\n"); // or: sb.toString().split()
```

Another StringBuilder example

```
StringBuilder sb = new StringBuilder();

sb.append("dog");
sb.append(" cat");
sb.insert(3, " hates");        // index 3, not word 3
System.out.println(sb.toString());
// Prints: dog hates cat

sb.reverse();
System.out.println(sb.toString());
// Prints: tac setah god
```

String Equality

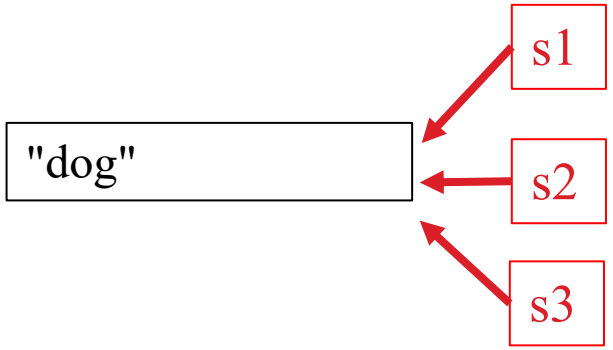
- String objects live on the heap, but their references live on the stack
  - Constant String objects have their own space
- Testing for equality with = = tests if the references are the same – do they point to the same thing on the heap?
- Normally, you want to ask, do these different String objects contain the same characters? Use `compareTo()` instead

String Equality

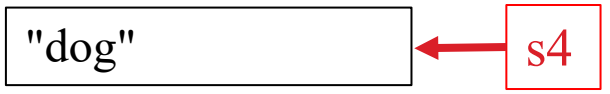
```
String s1 = "dog";
String s2 = "dog";
String s3 = s1;          // Copies reference, not the object

System.out.println(s1 == s2);        // true, but …
System.out.println(s1.equals(s2));   // true
System.out.println(s1 == s3);        // true, but …
System.out.println(s1.equals(s3));   // true

System.out.println("Enter a word:");
// Assume scanner was initialized; enter "dog"
String s4 = scanner.next();
System.out.println(s1 == s4);        // false
System.out.println(s1.equals(s4));   // true
```

Heap Allocation for String

Constant strings



Heap allocation



Wrapper Types

- Each primitive type as an associated ***wrapper class*** type that contains the primitive type as its member data item
  - `Boolean, Character, Byte, Short, Integer, Long, Float, Double, Void`
- These classes have useful methods – for example, to convert from String to int, use the `Integer` class `parseInt( )` method – but do ***not*** choose them for normal usage

Autoboxing and Unboxing

- One typical use case is when you need to store data inside a `Collection` (like an `ArrayList`). The contained data can only be of class types, not primitives, so you have to use a wrapper to convert.
- Autoboxing means wrapping the primitive type in a wrapper – can use methods, but not needed:

```
int i = 7;
Integer j = i; // Autobox
```

Wrapper Types, cont.

```
Integer myInt1 = new Integer(5);            // Using constructor
Integer myInt2 = new Integer(7);
Integer answer = Integer.sum(myInt1, myInt2);   // Okay, but why?

Double x = new Double( );
x = 7.3*0.2/1.9;                            // Autoboxing
double y = x;                               // Unboxing

String s = "123";
int value = Integer.parseInt(s);
// Or use Scanner's nextInt( ) method
System.out.println("value = " + value);
```

High-Precision Types

- To overcome the int and double limits, these classes were invented:
  - `BigInteger`: arbitrary precision integers
  - `BigDecimal`: fixed precision integers
- Both are significantly slower than primitive types
- Sample use cases:
  - encryption
  - accurate money

High Precision Types, cont.

Why String? Internally stored that way

```
BigInteger big = new BigInteger("1");
BigInteger big2 = new BigInteger("123456789012345");

// Add them together
BigInteger answer = big.add(big2);
System.out.println(answer);
// Prints: 123456789012346
```

Use add( ), not +