
COMP 322: Fundamentals of Parallel Programming

Lecture 5: Data and Control Flow in Async Tasks, Data Races

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

<https://wiki.rice.edu/confluence/display/PARPROG/COMP322>



Acknowledgments

- Chapter 4 of “The Art and Science of Java”, Eric Roberts, Stanford University
- CS201J Lecture 2: Java Semantics, David Evans
— www.cs.virginia.edu/cs201j/lectures/lecture2.ppt



Goals for Today's Lecture

- Understanding Data and Control Flow between an Async Task and its Parent
- Data Races and How to Avoid Them

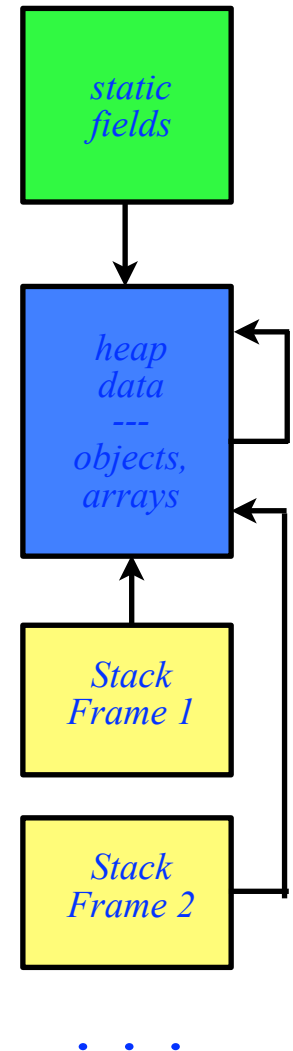


Recap of Java's Storage Model for Sequential Programs

Java's storage model contains three memory regions:

1. Static Data: region of memory reserved for variables that are not allocated or destroyed during a class' lifetime, such as static fields.
2. Stack Data: Each time you call a method, Java allocates a new block of memory called a stack frame to hold its local variables.
3. Heap Data: region of memory for dynamically allocated objects and arrays (created by "new").

All references (pointers) must point to heap data
--- no references can point to static or stack data

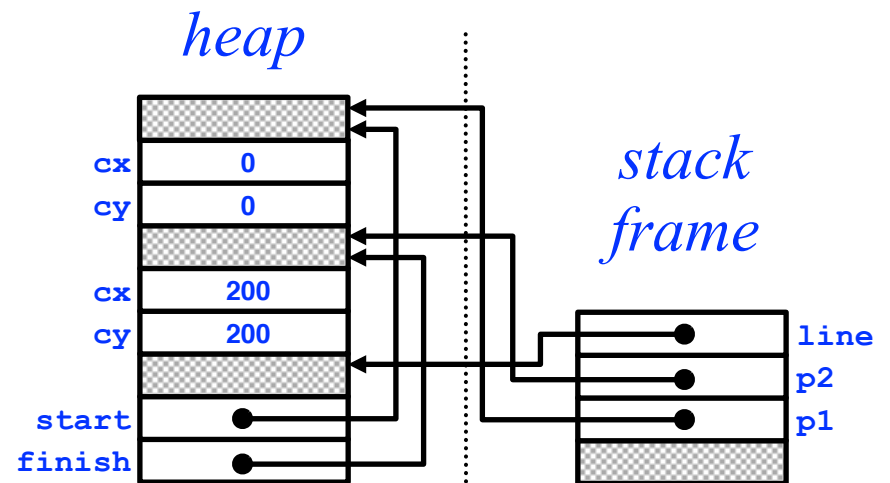


Example of Stack-to-Heap and Heap-to-Heap Pointers

```
public void run() {
    Point p1 = new Point(0, 0);
    Point p2 = new Point(200, 200);
    Line line = new Line(p1, p2); // Heap-stack diagram is for this stmt
}

public class Line {
    public Line(Point p1,
                Point p2) {
        start = p1;
        finish = p2;
    }
    ...
    private Point start;
    private Point finish;
}

public class Point {
    public Point(int x, int y) {
        cx = x;
        cy = y;
    }
    ...
    private int cx;
    private int cy;
}
```

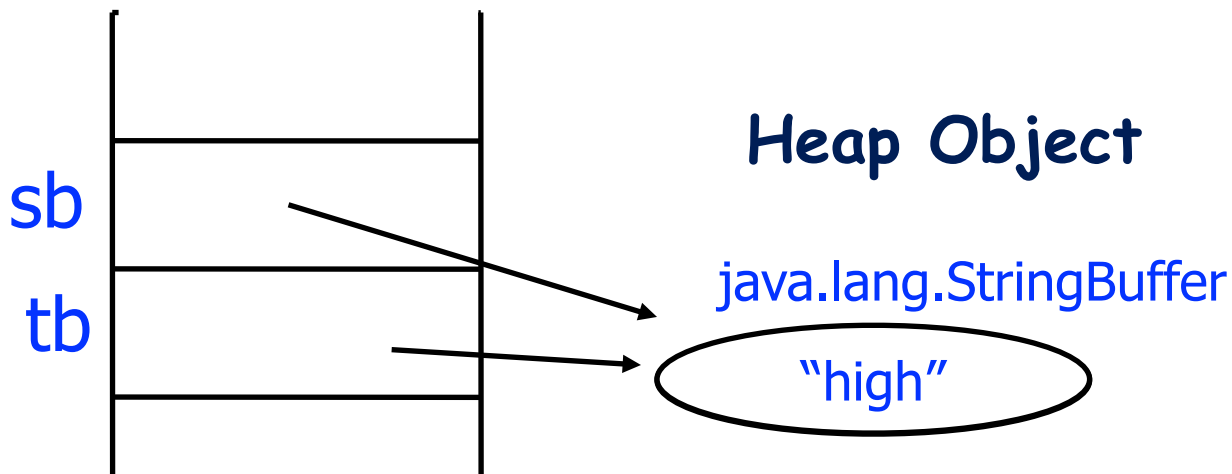


Mutability

- If an object is modified, all references to the object see the new value

```
StringBuffer sb = new ("hi");  
StringBuffer tb = sb;  
tb.append ("gh");
```

Stack Frame



Parameter Passing in Java

- Call-by-value: All parameters in Java are passed by value. The method receives a copy of the parameter, not the caller's local variable.
- Parameters can only contain primitives and references. Copying a reference (pointer) does not make a copy of the object pointed to.
- Caller and callee methods can communicate in the following ways
 - Parameters: callee receives a parameter from the caller in the form of a local variable (stack data)
 - Return values: callee can return a single value as a local variable for the caller (stack data)
 - Caller and callee can both read/write the same static fields (static data)
 - Caller and callee can both read/write the same objects and arrays (heap data)



Example: Use of Static Fields to Communicate Return Values from a Method (Poor Programming Practice)

```
1.  static int sum1 = 0, sum2 = 0;
2.  static void computeSum1Sum2(int[] X) { // callee
3.      for(int i=X.length/2; i < X.length; i++) sum2 += X[i];
4.      for(int i=0; i < X.length/2; i++) sum1 += X[i];
5.  }
6.  public static void main(String[] argv) { // caller
7.      int[] X = new int[...];
8.      ... // Initialize X
9.      int sum;
10.     computeSum1Sum2(X); // Call cannot update sum in main()
11.     sum = sum1 + sum2;
12.     ....
13. }
```



Example: Use of an Object to Communicate Return Values from a Method (Preferred Approach)

```
1.  public class TwoIntegers {int sum1; int sum2;}
2.  . . .
3.  static TwoIntegers computeSum1Sum2(int[] X) { // callee
4.      TwoIntegers r = new TwoIntegers();
5.      for(int i=X.length/2; i < X.length; i++) r.sum2 += X[i];
6.      for(int i=0; i < X.length/2; i++) r.sum1 += X[i];
7.      return r;
8.  }
9.  public static void main(String[] argv) { // caller
10.     int[] X = new int[...]; ... // Initialize X
11.     int sum;
12.     TwoIntegers s = computeSum1Sum2(X);
13.     sum = s.sum1 + s.sum2;
14.     ....
15. }
```



How can an Async Task interact with its Parent Task?

- **Data flow**
 - Async task can read from static fields, objects, arrays, and local variables written by parent task
 - Same rule as method calls, except that parent's local variables are passed as implicit parameters
 - Async task can write to static fields, objects, arrays (but not parent's local variables) to be read by parent task after end-finish
 - Same rule as method calls, except that method calls also have return values
 - We will learn soon about an extension to asyncs with return values (futures)
- **Control flow**
 - Async task can execute a return statement (different from method return)
 - Async task can throw an exception
 - NOTE: break/continue cannot cross async boundaries



Data Flow: Use of Static Fields to Communicate Return Value from an Async Tasks (Poor Programming Practice)

```
1.  static int sum1 = 0, sum2 = 0;
2.  public static void main(String[] argv) { // caller
3.      int[] X = new int[...];
4.      ... // Initialize X
5.      int sum;
6.      finish { // Async's have same access rules as methods
7.          async for(int i=X.length/2; i < X.length; i++)
8.              sum2 += X[i];
9.          async for(int i=0; i < X.length/2; i++)
10.             sum1 += X[i];
11.      }
12.      sum = sum1 + sum2;
13.      ....
14. }
```



Data Flow: Use of an Object to Communicate Return Values from Async Tasks (Preferred Approach)

```
1.  public class TwoIntegers {int sum1; int sum2;}
2.  . . .
3.  public static void main(String[] argv) { // caller
4.  int[] X = new int[...]; ... // Initialize X
5.  int sum;
6.  TwoIntegers r = new TwoIntegers();
7.  finish { // Async's have same access rules as methods
8.      async for(int i=X.length/2; i < X.length; i++)
9.          r.sum2 += X[i];
10.     async for(int i=0; i < X.length/2; i++)
11.         r.sum1 += X[i];
12.     }
13.     sum = r.sum1 + r.sum2;
14.     ....
15. }
```



Control Flow: Semantics of HJ return statement

- Java semantics for return
 - Return from enclosing method
- HJ semantics for return statement
 - Return from immediately enclosing async or method

```
1. void foo() {  
2.   if (...) return; // Returns from method foo()  
3.   async { ... return; ... } // Returns from async  
4.   . . .  
5. }
```



Control Flow: Semantics of HJ break and continue statements

- Java semantics for break/continue
 - Perform appropriate action for innermost enclosing loop (or labeled loop)
 - It's an error to execute a break/continue statement without an enclosing loop
- HJ semantics for break/continue
 - It's also an error to execute a break/continue statement in an async without an enclosing loop in the same async
 - Cryptic error message from HJ compiler
 - "Target of branch statement not found"

```
1. void foo() {  
2.   async {  
3.     while (...) { ... break; ... } // Okay  
4.     break; // Error  
5. }
```



Some Common Errors in Lab 2

```
1.  finish for (int i = 0; i <= N - M; i++) {
2.      int j;
3.      async {
4.          for (j = 0; j < M; j++) {
5.              async {
6.                  if (text[i+j] != pattern[j]) break;
7.              }
8.              if (j == M) return i; // found at offset i
9.          }
10. }
```



Some Common Errors in Lab 2

```
1.  finish for (int i = 0; i <= N - 1; i++) {
2.      int j;
3.      async {
4.          for (j = 0; j < M; j++) {
5.              async {
6.                  if (text[i+j] != pattern[j]) break;
7.              }
8.              if (j == M) return i; // found at offset i
9.          }
10. }
```

Async cannot
modify local variable in
parent's scope

No loop
enclosing break
in async

Return statement
in basic async task cannot take
a value



Async-Finish Exception Semantics

- Exceptions thrown by multiple `async`'s are accumulated into a “MultipleExceptions” collection at their Immediately Enclosing Finish

```
1.  try {
2.      finish for (int i = 0; i < size; i++)
3.          async {
4.              // Add explicit ArrayIndexOutOfBoundsException with X[-1]
5.              X[2*i*step] += X[(2*i+1)*step] + X[-1];
6.          } // finish-for-async
7.  } // try
8.  catch (Throwable t) {
9.      if (t instanceof MultipleExceptions)
10.         ... // Process the collection, t.exceptions
11.      else // single exception
12.         ... // Process t
13.  }
```



Goals for Today's Lecture

- Understanding Data and Control Flow between an Async Task and its Parent
- Data Races and How to Avoid Them



Example of Incorrect Parallelization from Homework 1

1. `// Sequential version`
2. `for (p = first; p != null; p = p.next) p.x = p.y + p.z;`
3. `for (p = first; p != null; p = p.next) sum += p.x;`
- 4.
5. `// Incorrect parallel version`
6. `for (p = first; p != null; p = p.next)`
7. `async p.x = p.y + p.z;`
8. `for (p = first; p != null; p = p.next)`
9. `sum += p.x;`

Why was this version incorrect?



Formal Definition of Data Races

Formally, a data race occurs on location L in a program execution with computation graph CG if there exist steps (nodes) $S1$ and $S2$ in CG such that:

1. $S1$ does not depend on $S2$ and $S2$ does not depend on $S1$ i.e., there is no path of dependence edges from $S1$ to $S2$ or from $S2$ to $S1$ in CG , and
2. Both $S1$ and $S2$ read or write L , and at least one of the accesses is a write.

Data races are challenging because of

- Nondeterminism: different executions of the parallel program with the same input may result in different outputs.
- Debugging and Testing: it is usually impossible to guarantee that all possible orderings of the accesses to a location will be encountered during program debugging and testing.



Five Observations related to Data Races

1. Immutability property: there cannot be a data race on shared immutable data.
 - A location, L , is immutable if it is only written during initialization, and can only be read after initialization. In this case, no read can potentially execute in parallel with the write.
- Parallel programming tip: use immutable objects and arrays to avoid data races
 - May require making copies of objects and arrays
 - Copying overhead may be prohibitive in some cases, but acceptable in others
- Example with `java.lang.String`

```
finish {  
    String s1 = "XYZ";  
    async { String s2 = s1.toLowerCase(); ... }  
    System.out.println(s1);  
}
```



Observations

2. Single-task ownership property: there cannot be a data race on a location that is only read or written by a single task.
- Define: step S in computation graph CG “owns” location L if S performs a read or write access on L . If step S belongs to Task T , we can also say that Task T owns L when executing S .
 - Consider a location L that is only owned by steps that belong to the same task, T . Since all steps in Task T must be connected by continue edges in CG , all reads and writes to L must be ordered by the dependences in CG . Therefore, no data race is possible on location L .



Avoiding Data Races: Copying for Single-task ownership

- If an object or array needs to be written multiple times after initialization, then try and restrict its ownership to a single task.
 - Entails making copies when sharing the object with other tasks.
 - As with Immutability, copying overhead may be prohibitive in some cases, but acceptable in others.

- **Example**

```
1. finish { // Task T1 owns A
2.   int[] A = new int[n]; // ... initialize array A ...
3.   // create a copy of array A in B
4.   int[] B = new int[A.length]; System.arraycopy(A,0,B,0,A.length);
5.   async { // Task T2 owns B
6.     int sum = computeSum(B,0,B.length-1); // Modifies B (ArraySum1 algorithm)
7.     System.out.println("sum = " + sum);
8.   }
9.   // ... update Array A ...
10.  System.out.println(Arrays.toString(A)); //printed by task T1
11.}
```



Observations (contd)

3. Ownership-transfer property: there cannot be a data race on a location if all steps that read or write it are totally ordered in *CG* (i.e., if the steps belong to a single directed path)
 - Think of the ownership of *L* being ``transferred'' from one step to another, even across task boundaries, as execution follows the path of dependence edges in the total order.
4. Local-variable ownership property: there cannot be a data race on a local variable.
 - If *L* is a local variable, it can only be written by the task in which it is declared (*L*'s owner). The copy-in semantics for local variables ensures that the value of the local variable is copied on async creation thus guaranteeing that there is no race condition between the read access in the descendant task and the write access in *L*'s owner.



Observations (contd)

5. Determinism property: if a parallel program with async and finish operations never exhibits a data race, then it must be deterministic with respect to its inputs.
- A computation is said to be “deterministic with respect to its inputs” if it always computes the same answer, when given the same inputs.
 - For the class of parallel programs that we have studied thus far, the absence of data races is sufficient to guarantee that the parallel program must be deterministic with respect to its inputs.
 - Such programs are said to be “data-race-free”. Programs that may exhibit data races are said to be “racy”.



Homework 2 Reminder

- Programming assignment, due Monday, Jan 30th
- Post questions on Piazza (preferred), or send email to comp322-staff at mailman.rice.edu
- You should plan to use turn-in script for HW2 submission
 - Contact teaching staff if you cannot access turn-in by following the instructions for Lab 1
- See course web site for penalties for late submissions

