ETH Zürich
Chair of Software Engineering

Java and C# in Depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Exercise Session – Week 9
GC and Managed Memory Leaks

Overview

Managed memory

- Why GC?
- GC overview
- Tricks & Memory Leaks
- Performance considerations

Pre-GC era

```
void Foo(int k){
 double* a = new double[k];
…
delete a;// memory leak! Plus, Undefined
      behavior (UB)
}
```

```
void Foo3(int k){
int* bar = new int[k];
…
delete[] bar;
..
delete[] bar; //UB
}
```

```
Bar* Create(){return new Bar;}
void Foo2(){
Bar* g = Create();
…//forgot to delete, memory leak!
}
```

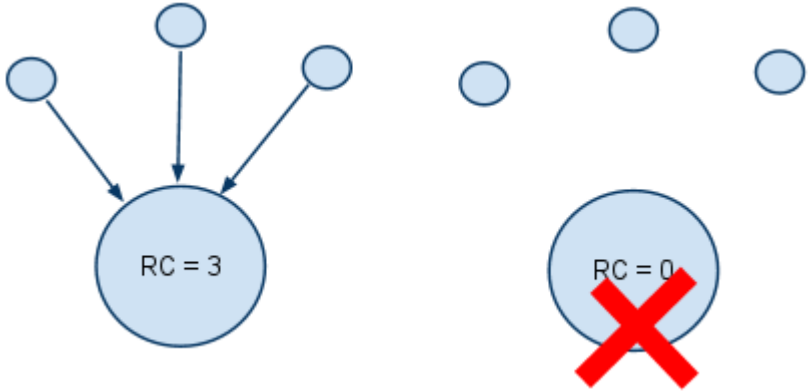
```
  void Foo4(){
   Bar* g = new Bar;
   …
   free(g); //UB, memory leak!
```

```
 void Foo5(){
 Bar* g = new Bar;
 delete g;
 g.crunch();//UB!
 }
```

Pre-GC era 2

Reference counting to the rescue!

```
Bar* Create(){..}
void BetterFoo(){
  std::shared_ptr<Bar> ref(Create());//1 ref
  {
     std::shared_ptr<Bar> ref2(ref)// 2 refs
  } // 1 ref
  ..
}//0 refs, Bar deleted
```



Is this enough?

Cyclic references

```
void FooNode(){
  Node *a = new Node; Node *b = new Node;
  a->Next = new std::shared_ptr(b);
  b->Next = new std::shared_ptr(a);
..
}
```

Even if there is no references outside to a, or b, they still cannot be deleted!

Need another approach!

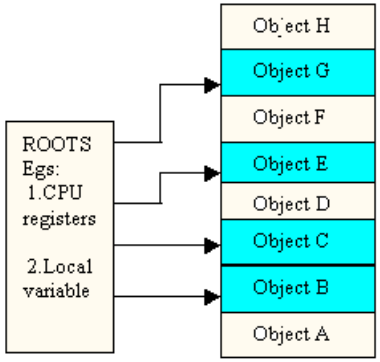
GC essentials: Mark & Compact

Stage 0: Stop all running threads

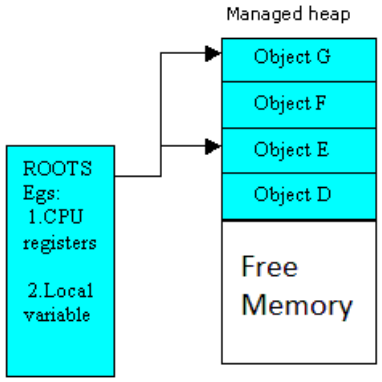
Stage 1: Mark reachable objects from the roots

What are roots?

- All local (stack) variables in all threads
- Static variables
- Content of CPU registers
- etc



Mark



Compact

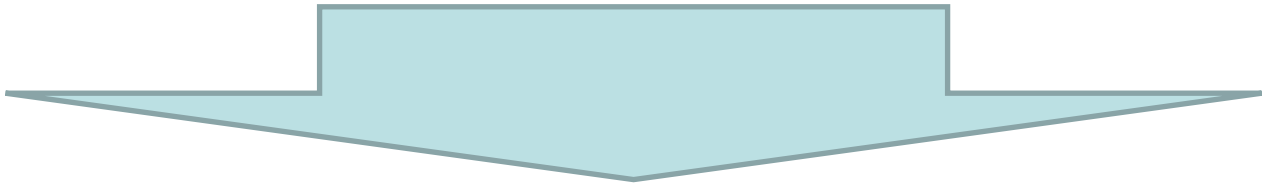
Stage 2: Compact everything that is marked. Everything else is garbage (no way to reach it)

GC: finalization

File f = new File(…); //consumes the machine resources

How does a framework know that it needs to close a file?

Garbage collection is non-deterministic!



Need to finalize some objects

- C# : ~File(){..}
- Java: protected void finalize(){…}

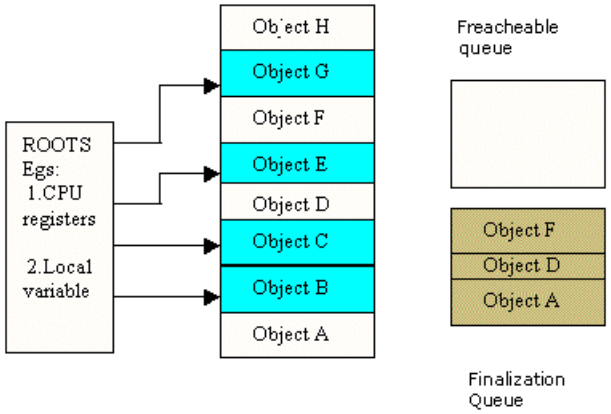
GC: Mark & Compact Revised(C#)

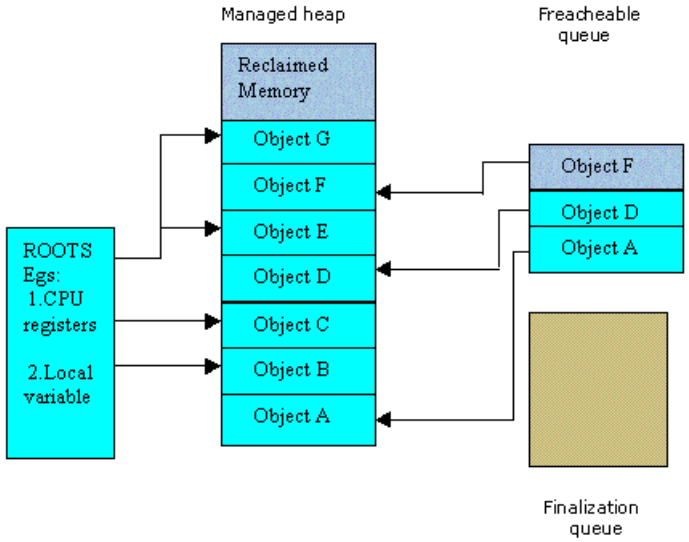
Stage 0: Stop all running threads

Stage 1: Mark reachable objects from the roots. Put the finalizable objects to Freachable (finalizer-reachable queue), also mark them

Stage 2: Compact everything that is marked. Everything else is garbage (no way to reach it)

Finalizer thread: Maintains the finalization queue, runs object finalizers in own thread. May run in parallel to actual program execution



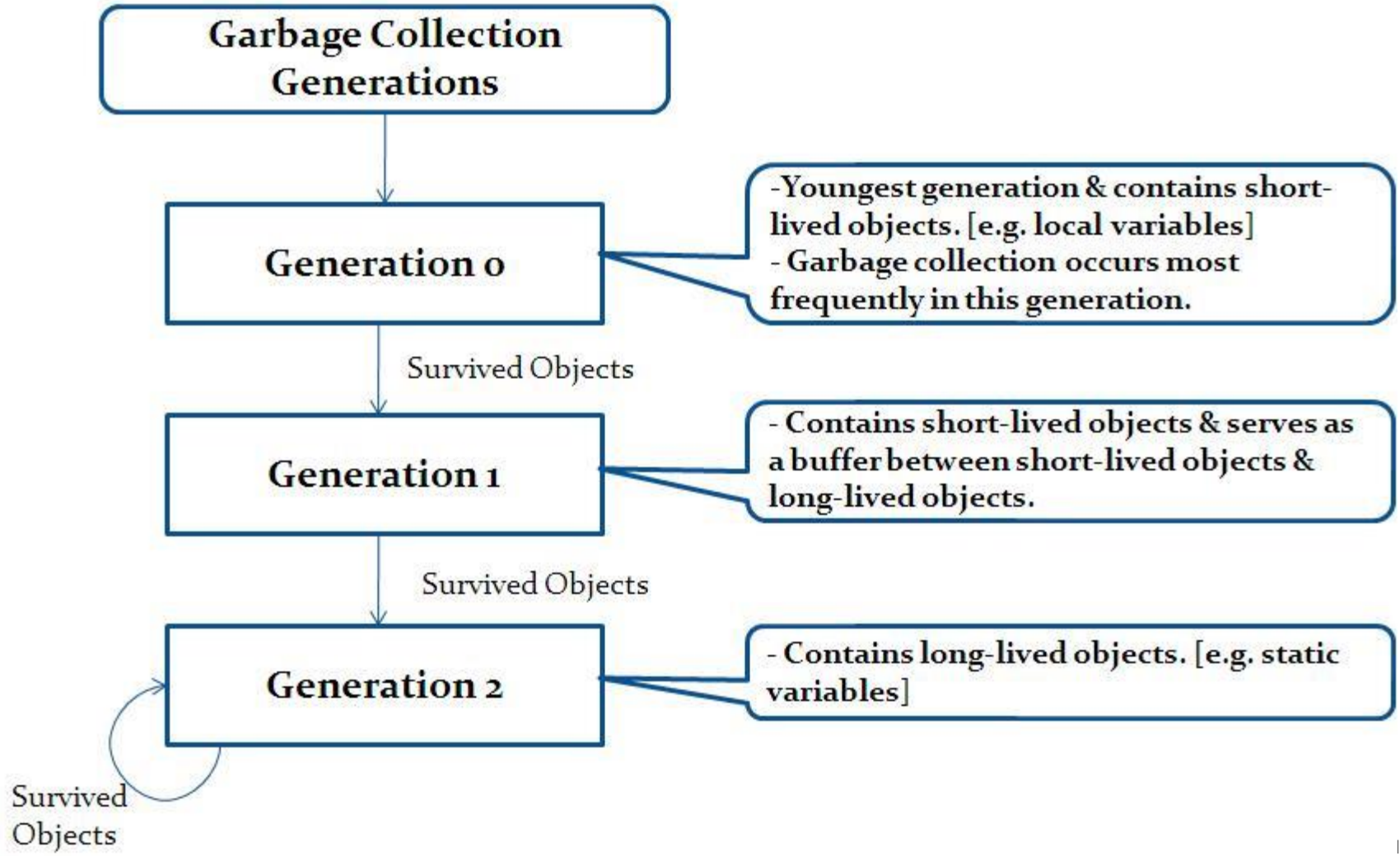


Consequences:

- Finalizable objects take longer to be collected => do not use without necessity

GC: generations

Optimization technique: do not process the entire heap, until necessary



GC: Non-determinism

SqlConnection conn = new SqlConnection(…);

~SqlConnection { //cleanup unmanaged resources, close the physical connection }

- When will connection be closed?
- Is it guaranteed to close at all?

"GC is when the operating environment automatically reclaims memory that is no longer being used by the program. It does this by tracing memory starting from roots to identify which objects are accessible"

The job of a firefighter is "driving a red truck and spraying water."

Insights:

- If the amount of RAM available to the runtime is greater than the amount of memory required by a program, then a memory manager which employs the null garbage collector is a valid memory manager.
- A correctly-written program cannot assume that finalizers **will ever run** at any point prior to program termination.

GC: Disposable pattern(C#)

interface IDisposable{ void Dispose();} - Deterministic finalization

For unmanaged resource holders – Disposable pattern:

```
class Holder : IDisposable{
    //some unmanaged resource
    ~Holder(){
        Dispose(false);
    }
    protected virtual void  Dispose(bool disposing){// if class is sealed, this method is private
        if(disposing){
          //managed cleanup
        }
        //free unmanaged resources: close file handlers, db connections, free unmanaged memory
    }
    public void Dispose(){
        Dispose(true);
        GC.SupressFinalize(this);
    }
}
```

GC: managed memory leak -1

```
class Processor
{
      MailManager _bar;
      public Processor(MailManager bar) {
        _bar = bar;
        _bar.NewMail += OnNewMail;
      }
      void OnNewMail(object sender, EventArgs e) { }
      public void DoSmth(){..}
 }
```

```
var bar = new MailManager();
while(!bar.Stopped)
{
    var f = new Processor(bar);
    f.DoSmth();
}
```

```
class MailManager
{
      public event EventHandler NewMail;
     //other methods
 }
```

Memory leak because of the event subscription.

Ways to solve:

- Unsubscribe, when object is not needed
- WPF WeakEventManager
- Custom Weak Events implementation

GC: managed memory leak - 2

A snippet from server application:

```
while(!IsStopped) {
  var data = (Foo)HttpContext.Current.Session[“userData"]
  LocalDataStoreSlot myData;
  myData=Thread.GetNamedDataSlot(“userData”);
  Thread.SetData(myData, data);
   //continue to listen
}
```

```
while(!IsStopped){
 var data = new Foo ();
dict.add (“userData”, data);
}
```

Thread Local Storage: continues to keep data, until it is explicitly cleared. Can be thought as static dictionary.

Again, the problem is that objects live longer, than needed – GC is **unable to help** with that!

How to solve? Thread.SetData(myData, null), when finished with processing

GC: enough memory?(C#)

Large Object Heap (LOH) stores the objects:

- sizeof(object) >= 85 kb
- double[], with size >= 1000

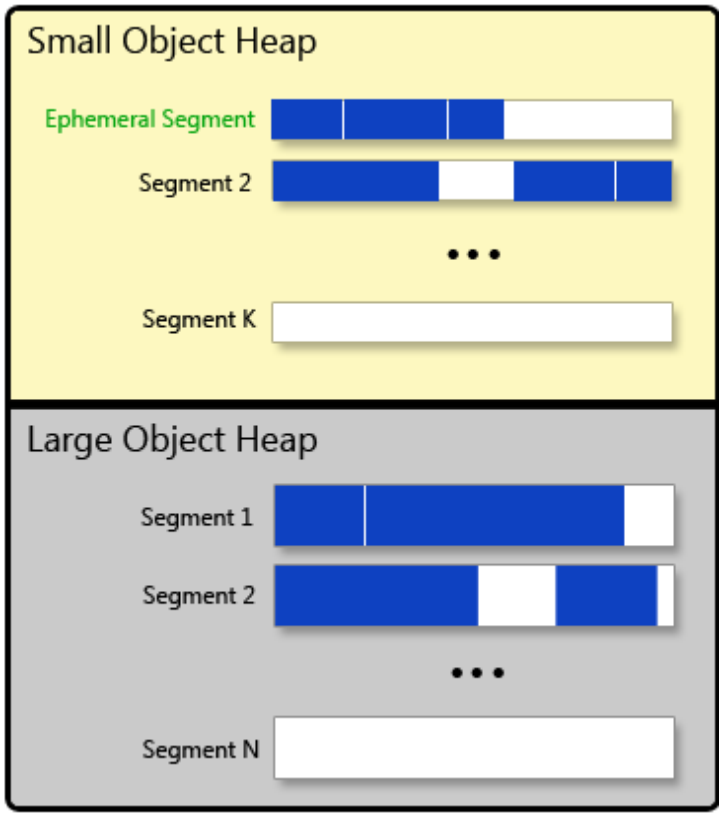
85k – threshold, when compacting does not provide any performance improvements

Memory in LOH does not get compacted =>

May end with OutOfMemoryException, even if there is enough memory in LOH's fragments

Insight:

- Use large objects with care



GC: out of scope

- Object resurrection
- Unmanaged memory leaks ☺
- Critical Finalization
- Weak References
- Interaction with unmanaged memory(memory pins)
- GC modes (server vs client)
- Debugging memory leaks(windbg, perfmon)

References

- GC explained : http://blogs.msdn.com/b/oldnewthing/archive/2010/08/09/10047586.aspx
- J. Richter weak events: http://wintellect.com/blogs/jeffreyr/weak-event-handlers
- Disposable pattern: http://msdn.microsoft.com/en-us/library/fs2xkftw.aspx
- Large Object Heap: http://msdn.microsoft.com/en-us/magazine/cc534993.aspx