Object-Oriented Languages

Chapters 7 and 9
Outline

• Classes and objects
• Methods
  – Inheritance, polymorphism
  – Static methods and fields
• Implementation: compilation, allocation
• Types
• Memory and type safety
• Memory management
Classes

• A **class** is a blueprint for creating objects

```java
class Rectangle {
    public double height, width;
    public double area() {
        return height * width;
    }
}
```

– This is Java code; the equivalent C++ code is very similar

• **Class members**: *methods* and *fields*
Objects

• The central concept of object-oriented programming

• In C++ and Java, they are instances of classes, created through new
  – E.g., when expression new Rectangle() is evaluated, a new object of class Rectangle is created and initialized
  – “instance” = “object”
  – “class X is instantiated” = “an instance of X is created”
References in Java/Pointers in C++

- Objects are manipulated indirectly through object references (pointers)

```java
main(...) { // Java code
    Rectangle x;
    x = new Rectangle(); // 1) Create a Rectangle object in memory
    // 2) Produce a reference value which is
    // a handle to this object
    // 3) Assign this reference value to x
    x.width = 3.14; // 1) Use the r-value of x to get to the object
    // 2) Assign based on the l-value of field width
}
```

- `x` is a variable of type “reference to Rectangle objects”

- C++: `Rectangle* x; x = new Rectangle();`
- `x` is a variable of type “pointer to Rectangle objects”
Creation of Objects

• During the evaluation of \( x = \text{new Rectangle}() \)
  – A new instance (object) of class Rectangle is created on the heap
  – A reference (pointer) to this instance is produced
    • This is the result of evaluating the \texttt{new} expression
  – The appropriate \texttt{constructor} of the class is called to initialize the new object
  – \( x \) is assigned this reference (pointer) value
    • e.g. the value may be the \texttt{address} of the first byte of the object’s memory
    • or the value may be some \texttt{internal handle} to the actual object (e.g., index in some internal table, which itself contains the address of the first byte)
Destruction of Objects

• C++: each new must have a corresponding delete
  – `x = new Rectangle(); ... delete x;`

• Java: dead objects are reclaimed automatically by a garbage collector (GC)
  – `x = new Rectangle(); // after you stop using the object, GC may figure out it is dead`

• C++ destructors: called when the programmer manually destroys the object with delete
  – `class Rectangle { ... ~Rectangle() {...} // destructor }`

• Java finalizers: called when the object is collected
  – `class Rectangle { ... void finalize() {...} // finalizer }`
Members: Fields and Methods

• Two separate kinds: **instance** members and **static** members
  
  – Instance members: each instance of the class has a separate copy of this member

```
Rectangle a, b, c;
a = new Rectangle();
b = new Rectangle();
a.height = 1.0; a.width = 3.6;
b.height = 2.2; b.width = 5.0;
c = a;
```
Members: Fields and Methods (C++)

- C++: \texttt{x->f} is shorthand (syntactic sugar) for \texttt{(*x).f}
  - Expression \texttt{x} evaluates to pointer value that points to the object; expression \texttt{*x} evaluates to the actual object; \texttt{*x->f} evaluates to the field \texttt{f} of that object (\texttt{f} is \textbf{not} static – why?)

Rectangle *a, *b, *c;

a = new Rectangle();

b = new Rectangle();

a->height = 1.0; a->width = 3.6;

b->height = 2.2; b->width = 5.0;

c = a;
Outline

• Classes and objects
• **Methods**
  – Inheritance, polymorphism
  – Static methods and fields
• Implementation: compilation, allocation
• Types
• Memory and type safety
• Garbage collection
Instance Methods

• An instance method operates on objects
  – Method \texttt{m} is invoked on the object

\begin{verbatim}
double \texttt{area()} { return height*width; }
\end{verbatim}

in reality, this is syntactic sugar for

\begin{verbatim}
double \texttt{area(Rectangle this) \{ // Java
    return this.height * this.width; \}}
double \texttt{area(Rectangle* this) \{ // C++
    return this-&gt;height * this-&gt;width; \}}
\end{verbatim}

• There is an implicit formal parameter \texttt{this}: a reference to the object on which the method was invoked
  – Calls \texttt{x.area()} and \texttt{x-&gt;area()} are, in essence, calls \texttt{area(x)}
Methods Calls

• Calling an instance method: there is an object on which we are calling it
  – x.m() in Java, x->m() in C++

Rectangle a, b, c;
a = new Rectangle();
a.height = 1.0;
a.width = 3.6;
c = a;
double result = c.area();
Constructors

• Constructors are used to set up the initial state of new objects

```java
public Rectangle(double height, double width) {
    this.height = height; this.width = width;
}
```

• `x = new Rectangle(1.1, 2.3);`
  – A new object is created: with default values 0.0 in Java, and undefined values in C++
    • The constructor is invoked on this object; the fields are initialized with 1.1 and 2.3
  – A reference to the object is assigned to `x`
Outline

• Classes and objects
• Methods
  – Inheritance, polymorphism
  – Static methods and fields
• Implementation: compilation, allocation
• Types
• Memory and type safety
• Memory management
Inheritance

• class B extends A { ... }
  – Single inheritance: only one superclass (Java)

• class B : public A1, A2, A3 { ... }
  – Multiple inheritance: several superclasses (C++)

• Every member of A is inherited by B
  – If a field f is defined in A, every object of class B has an f field
  – If a method m is defined in A, this method can be invoked on an object of class B

• B may declare new members
Example

class Rectangle {
    private double height, width;
    public Rectangle(double h, double w) { ... }
    public double getHeight() { return height; }
    public double getWidth() { return width; }
    public double area() { ... }
}

class SwissRectangle extends Rectangle {
    private int hole_size;
    public SwissRectangle(double h, double w, int hs) { ... }
    public void shrinkHole() { hole_size--; }
    public double area() { ... } // overridden
}
Constructors and Inheritance

• Constructors are not inherited
• A constructor in a subclass B must invoke a constructor in the superclass A
  – (this is a bit of an oversimplification)
• The constructor of superclass A initializes the part of the “object state” that is declared in A
  – Sets up values for fields declared in A and inherited by the subclasses

```java
class SwissRectangle extends Rectangle {
    private int hole_size;
    public SwissRectangle(double h, double w, int hs) {
        super(h,w); hole_size = hs; }
```
Inheritance of Methods

• If a subclass declares a method with the same name but **a different signature**, we have **overloading**
  – Either method can be invoked on an instance of the subclass

• If a subclass declares a method with **the same signature**, we have **overriding**
  – Only the new method applies to instances of the subclass
Polymorphism of References

• Reference variables for A objects also may points to B objects
  – A x = new B() in Java; A* x = new B() in C++

• Simplistic view: the type of x is pointer (reference) to instances of A

• Correct view: pointer to instances of A or instances of any subclass of A
  – If C is a subclass of B, variable x can also point to instances of C
  – Poly (many) morph (form) ism
Method Invocation – Compile Time

- What happens when we have a method invocation of the form \texttt{x.m(a,b)}?
- Two very different things are done
  - At \texttt{compile time}, by the Java compiler (javac)
  - At \texttt{run time}, by the Java Virtual Machine
- At compile time, a target method is associated with the invocation expression
  - Terms: \texttt{compile-time target}, \texttt{static target}
  - The static target is based on the \texttt{declared type of x}
Method Invocation – Compile Time

class A { void m(int p, int q) {...} ... }
class B extends A { void m(int r, int s) {...} ... }

A x;
x = new B();
x.m(1,2);

• Since x has declared type A, the compile-time target is method m in class A

• javac encodes this in the bytecode (classname.class)
  • virtualinvoke x,<A: void m(int,int)>
Method Invocation – Run Time

• The Java virtual machine loads the bytecode and starts executing it

• When it tries to execute instruction `virtualinvoke x,<A: void m(int,int)>`
  – Looks at the class Z of the object referenced by x
  – Searches Z for a method with signature `m(int,int)` and return type `void`
  – If Z does not have it, goes to Z’s superclass, and so on upwards, until a match is found
Method Invocation – Run Time

• The **run-time (dynamic) target**: “lowest” method that matches the signature and the return type of the static target method
  – “Lowest” with respect to the inheritance chain from `Z` to `java.lang.Object`
• Once the JVM determines the run-time target method, it invokes it on the object that is referenced by `x`
• Terms: **virtual dispatch, method lookup**
Virtual Methods in C++

class A { virtual void m(int p, int q) {...} ... }
class B : public A
    { virtual void m(int r, int s) {...} ... }

A* x;
x = new B();
x->m(1,2);

• Since x has declared type A*, the compile-time target is method m in class A

• The run-time target is m in B
  • Without the keyword virtual, the run-time target will be the same as the compile-time target
Abstract Classes

• Abstract class: class that contains abstract methods
  – `abstract void m(int x);` // Java
  – `virtual void m(int x) = 0;` // C++

• We cannot say `new X()` if X is abstract. Why?

• An abstract method can be the **compile-time target** of a method call
  – But not the run-time target, obviously

• Sometimes non-abstract classes are referred to as “concrete classes”
Interfaces in Java

- Very similar to abstract classes in which all methods are abstract
- A Java class has only one superclass, but can implement many interfaces
  - `class Y extends X implements A, B { ... }`
- A reference variable can be of interface type, and can refer to any instance of a class that implements the interface
- An interface method can be the `compile-time target` of a method call
Example

interface X { void m(); }
interface Y { void n(); }
abstract class A implements X {
    void m() { ... }
    abstract void m2();
}
class B extends A implements Y {
    void m2() { ... }
    void n() {...}
}

X x = new B(); x.m();
Y y = new B(); y.n();
A a = new B(); a.m2();
Outline

• Classes and objects
• Methods
  – Overloading, constructors
  – Inheritance, polymorphism
  – Static methods and fields
• Implementation: compilation, allocation
• Types
• Memory and type safety
• Memory management
Static Methods and Fields

• **Static field**: a single copy for the entire class

• **Static method**: *not* invoked on an object
  – Just like a regular procedure (function) in a procedural language (e.g., C, Pascal, etc.)

• Terminology
  – static method/field = **class** method/field
  – instance method/field = **non-static** method/field
class X { ...

    private static int num = 0;

    // constructor
    public X() { num++; }

    public static int numInstances()
    {
        return num;
    }

}  

in main:

X x1 = new X(); X x2 = new X();
int n = X.numInstances();  returns 2
Static Example (C++)

class X { ...

    private: static int num;
    public: X();
    public: static int numInstances();

} } 

int X::num = 0;
X::X() { num++; } 

int X::numInstances() { return num; } 

in main:
X* x1 = new X; X* x2 = new X;
int num = X::numInstances(); returns 2
Example: Singleton Pattern (Java)

class Logger {
    private Logger() { }
    private static Logger instance = null;
    public static Logger getInstance() {
        if (instance == null)
            instance = new Logger();
        return instance;
    }
}

client code: Logger.getInstance().writeLog(…)

Outline

• Classes and objects
• Methods
  – Overloading, constructors
  – Inheritance, polymorphism
  – Static methods and fields
• **Implementation: compilation, allocation**
• Types
• Memory and type safety
• Memory management
Implementation Techniques for Java

• The compiler takes as input source code
  – Oracle/Sun provides a standard compiler; others can build their own compilers if they want
  – Typically, class A is stored in file A.java
    • Exception: nested classes

• Compiler output: Java bytecode
  – A.java -> A.class
  – A standardized platform-independent representation of Java code
  – Essentially, a programming language that is understood by the Java Virtual Machine
Rectangle.class

class Rectangle extends java.lang.Object {
    public double height;  public double width;
    Rectangle();
    public double area();
}

Rectangle()
    0 aload_0
    1 invokespecial #3 <Method java.lang.Object()>
    4 return
double area()
    0 aload_0
    1 getfield #4 <Field double height>
    4 aload_0
    5 getfield #5 <Field double width>
    8 dmul
    9 dreturn
Execution Model

• Java bytecode is executed by a Java Virtual Machine (JVM)
  – Oracle/Sun provides several kinds of JVMs for various platforms (e.g., Solaris, Wintel, etc.)
  – Several other vendors for JVMs
    • E.g., IBM sells a JVM that is performance-tuned for enterprise server applications

• Platform independence: as long as there are JVMs available, the exact same Java bytecode can be executed anywhere
JVM

• There are two ways to execute the bytecode

• **Interpretation**: the VM just executes each bytecode instruction itself
  – Initial JVMs used this model

• **Compilation**: the VM uses its own internal compiler to translate bytecode to native code for the platform
  – The native code is executed by the platform
  – Faster than interpretation
Compilation Inside a VM

• **Just-in-time**: the first time some bytecode needs to be executed, it is compiled to native code on the fly
  – Typically done at method level: the first time a method is invoked, the compiler kicks in
  – Problems: compilation has overhead, and the overall running time may actually increase

• **Profile-driven** compilation
  – Start executing through interpretation, but track “hot spots” (e.g., frequently executed methods), and after a certain threshold is reached, point compile them
Lifetimes and Memory Management

• **Static allocation**: address determined once and retained throughout the execution of the program
  – E.g., *static* fields in C++, Java

• **Stack-based allocation**: local variables of methods, plus the formal parameters (incl. *this*)

• **Heap-based allocation**: space allocated and deallocated manually by the programmer
  – C: `A* a = (A*)malloc(sizeof(A)); ... free (a);`
  – C++: `A* a = new A(); ... delete a;`
  – Java: `A a = new A();` but deallocation is performed automatically
Outline

• Classes and objects
• Methods
  – Overloading, constructors
  – Inheritance, polymorphism
  – Static methods and fields
• Implementation: compilation, allocation
• Types
• Memory and type safety
• Memory management
Types

• Organization of untyped values
  – Untyped universes: bit strings, S-expr, ...
  – Categorize based on usage and behavior

• Type = set of computational entities with uniform behavior

• Constraints to enforce correctness
  – Check the applicability of operations
    • Should not try to multiply two strings
    • Should not use a character value as a condition of an if-statement
    • Should not use an integer as a pointer
Examples of Type Checking

• Built-in operators should get operands of correct types
• Type of left-hand side must agree with the value on the right-hand side
• Procedure calls: check number and type of actual arguments
• Return type should match returned value
Static Typing

- Statically typed languages: expressions in the code have **static types**
  - static type = claim about run-time values
  - Types are either **declared** or **inferred**
  - Examples: C, C++, Java, ML, Pascal, Modula-3

- A statically typed language typically does some form of **static type checking**
  - E.g., at compile time Java checks that the [] operator is applied to a value of type “array”
  - May also do dynamic (run-time) checking
    - e.g., Java checks at run time for array indices out of bounds and for null pointers
Dynamic Typing

• Dynamically-typed languages: entities in the code do not have static types
  – Examples: Lisp, Scheme, CLOS, Smalltalk, Perl, Python
  – Entities in the code do not have declared types, and the compiler does not try to infer types for them

• Dynamic type checking
  – Before an operation is performed at run time
  – E.g., in Scheme: (+ 5 #t) fails at run time, when the evaluation expects to see two numeric atoms as operands of +
Strongly vs. Weakly Typed

- **Strongly typed** languages: type-incorrect operations are not performed at run time
  - Things cannot “go wrong”: no undetected type errors
  - Certain run-time errors are possible but clearly marked as such
    - i.e. array index out of bounds, null pointer
  - C/C++: weakly typed
  - Java: strongly typed

- **Independent of static vs. dynamic**
  - Lisp, Scheme, Python: strongly, dynamically typed
  - Forth: weakly, dynamically typed
Examples of Types

• Integers
• Arrays of integers
• Pointers to integers
• Records with fields \texttt{int x} and \texttt{int y}
  – e.g., “struct” in C
• Objects of class C or a subclass of C
  – e.g., C++, Java, C#
• Functions from any list to integers
Numeric Types

• Varied from language to language
• C does not specify the ranges of numeric types
  – Integer types: char, short, int, long, long long
    • Includes “unsigned” versions of these
  – Floating-point types: float, double, long double
• Java specifies the ranges of numeric types
  – byte: 8-bit signed two's complement integer [-128, +127]
  – short: 16-bit signed two's complement integer [-32768, +32,767]
  – int: 32-bit signed two's complement integer
    [-2147483648, +2147483647]
  – long: 64-bit signed two's complement integer
    [-9223372036854775808, +9223372036854775807]
  – float/double: single/double-precision 32-bit IEEE 754 floating point
  – char: single 16-bit Unicode character; minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65535)
Enumeration Types

• C: a set of named integer constant values
  – Example from the C specification
    
    ```c
    enum hue { chartreuse, burgundy, claret=20, winedark };
    /* the set of integer constant values is { 0, 1, 20, 21 } */
    enum hue col, *cp;
    col = claret; cp = &col;
    if (*cp != burgundy) ...
    ```

• Java: a fixed set of named items (not integers)
  
    ```java
    enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }
    
    – In reality, it is like a class: e.g., it can contain methods
    ```
Types as Sets of Values

- **Integers**
  - Any number than can be represented in 32 bits in signed two’s-complement
  - "type \texttt{int}" = \{-2^{31}, \ldots, 2^{31} - 1 \}
- **Class type (not the same as a class)**
  - Any object of class \texttt{C} or a subclass of \texttt{C}
  - "type \texttt{C}" = set of all instances of \texttt{C} or of any transitive subclass of \texttt{C} ("class \texttt{C}" is just a blueprint for objects)
- **Subtypes are subsets**: \texttt{T2} is a **subtype** of \texttt{T1} if \texttt{T2}'s set of values is a subset of \texttt{T1}'s set of values
Monomorphism vs. Polymorphism

• Greek:
  – mono = single
  – poly = many
  – morph = form

• Monomorphism
  – Every computational entity belongs to exactly one type

• Polymorphism
  – A computational entity can belong to multiple types
Types of Polymorphism

- Parametric
- Universal
  - Inclusion (subset)
- Ad hoc
  - Overloading
  - Coercion
Types of Polymorphism

- parametric
- universal
- inclusion (subset)
- overloading
- ad hoc
- coercion
Coercion

• Values of one type are silently converted to another type
  – e.g. addition: 3.0 + 4: converts 4 to 4.0
    • \texttt{int} \times \texttt{int} \rightarrow \texttt{int} or \texttt{real} \times \texttt{real} \rightarrow \texttt{real}

• In a context where the type of an expression is not appropriate
  – either an automatic coercion (conversion) to another type is performed automatically
  – or if not possible: compile-time error
Coercions

• Widening
  – coercing a value into a “larger” type
  – e.g., `int` to `float`, subclass to superclass

• Narrowing
  – coercing a value into a “smaller” type
  – loses information, e.g., `float` to `int`
Types of Polymorphism

- Parametric
- Universal
  - Inclusion (subset)
- Overloading
  - (Operators or functions)
- Ad hoc
  - Coercion
Types of Polymorphism

- parametric
- universal
  - inclusion (subset)
  - overloading
  - coercion
- ad hoc
- polymorphism
package java.util;

public interface Set<E> extends Collection<E> { ...
    Iterator<E> iterator();
    boolean add(E e);
    boolean addAll(Collection<? extends E> c); }

class Rectangle { ...

class SwissRectangle extends Rectangle { ...

Set<Rectangle> s = new HashSet<Rectangle>();
s.add(new Rectangle(1.,2.)); s.add(new SwissRectangle(3.,4.,5));
Set<SwissRectangle> s2 = new TreeSet<SwissRectangle>();
s2.add(new SwissRectangle(6.,7.,8)); s.addAll(s2);
Types of Polymorphism

- parametric
- inclusion (subset)
- overloading
- coercion

- universal

- ad hoc

- polymorphism
Inclusion (Subset) Polymorphism

• Subtype relationships among types
  – Defined by “Y is subset of X” (i.e., set inclusion)
• A computational entity of a subtype may be used in any context that expects an entity of a supertype
• Typical examples
  – Imperative languages: record types
  – Object-oriented languages: class types
Subtyping in Java

• Recall that **class type C** is the set of all instances of class C or of any transitive subclass of C

• Subtyping between class types
  
  ```
  class X {
    int m () { ... }
  }
  class Y extends X {
    int m () { ... }
  }
  X x = new Y();
  int i = x.m();
  ```

• Interface type: the set of all instances of classes that implements the interface (transitively)
  
  ```
  interface Z {
    bool m();
  }
  class W implements Z {
    bool m() { ... }
  }
  Z z = new W();  bool b = z.m();
  ```
Outline

• Classes and objects
• Methods
  – Overloading, constructors
  – Inheritance, polymorphism
  – Static methods and fields
• Implementation: compilation, allocation
• Types
• **Memory and type safety**
• Memory management
Memory and type safety

```java
int[] a = new int[64];
...
a[82] = ...;
```
int[] a = new int[64];
...
a[82] = ...;

C: undefined behavior → major source of security exploits (how?)

Java: throws ArrayIndexOutOfBoundsException
How? Instrumentation added by JIT compiler (or proved unnecessary)
Memory and type safety

MyType* a = new MyType();

... 

*(((void*)a + 16) = 42;
Memory and type safety

MyType* a = new MyType();
...
*((void*)a + 16) = 42;

C++: Undefined behavior

Java: Pointer arithmetic isn’t part of the language
Memory and type safety

SomeType* a = ...;

b = (IncompatibleType*) a;
b.f = ...;
Memory and type safety

SomeType* a = ...;

b = (IncompatibleType*) a;
b.f = ...;

C: undefined behavior $\rightarrow$ potential security exploit

Java: throws ClassCastException

How? Instrumentation added by JIT compiler (or proves unnecessary)
Memory and type safety

MyType* a = new MyType(); /* or: malloc(sizeof(MyType)) */
...
delete a; /* or: free(a) */
...
a->f = ...;
Memory and type safety

MyType* a = new MyType();

...

delete a;

...

a->f = ...;
Memory and type safety

MyType* a = new MyType();

... delete a;

... a->f = ...;

C++: Undefined behavior

Java: Garbage collection → no explicit freeing
Memory and type safety

MyType* a = new MyType();

... delete a;

... delete a;
Memory and type safety

MyType* a = new MyType();
...
delete a;
...
delete a;

C++: Undefined behavior

Java: Garbage collection → no explicit freeing
Memory and type safety

MyType* a = new MyType();
container->data = a;

... container->data = NULL; /* Last ptr to a is lost */
Memory and type safety

MyType* a = new MyType();
container->data = a;
...
container->data = NULL; /* Last ptr to a is lost */

C++: Memory leak

Java: Garbage collection
How? Knows all reference types and all “roots”; approximates liveness via transitive reachability
Outline

• Classes and objects
• Methods
  – Overloading, constructors
  – Inheritance, polymorphism
  – Static methods and fields
• Implementation: compilation, allocation
• Types
• Memory and type safety
• Memory management
Live and dead objects

- **Live** object will be used in the future
- Other objects are **dead**
- Deallocate as soon as possible after last use (but not before!)

Memory management: Deallocate the **dead** objects in a “timely” fashion
Explicit (manual) memory management

• More code to maintain
• Requires global reasoning
• Correctness
  – Free a live object
  – Free a dead object too late (or never)
• Efficiency can be very high
  – Gives programmers more control over the run-time behavior of the program
Automatic memory management (garbage collection)

• Integral for memory and type safety
  – Protects against some classes of memory errors (dangling pointers, double frees)
  – Essential for Java, C#, PHP, JavaScript, ...
• Reduces programmer burden
• Not perfect, memory can still leak
  – Programmers still need to eliminate all pointers to objects the program no longer needs
• (A mostly solved) challenge: performance
What is Garbage?

• In theory, any object the program will never reference again
  – But compiler & runtime system cannot figure that out
• In practice, any object the program cannot reach is garbage
  – Approximate liveness with reachability
• Managed languages couple GC with “safe” pointers
  – Programs may not access arbitrary addresses in memory (e.g., Java/C# vs. C/C++)
  – The compiler can identify and provide to the garbage collector all the pointers, thus enforcing “Once garbage, always garbage”
  – Runtime system can move objects by updating pointers
Liveness approximates reachability

Diagram:
- Live
- Reachable
- Dead
- Unreachable
Liveness approximates reachability
• Can leaks happen in GC’d languages?
• Can leaks happen in GC’d languages?
• Can leaks happen in GC’d languages?
• Can leaks happen in GC’d languages?
Memory leak example

• Driverless truck
  – 10,000 lines of C#
• Leak: past obstacles remained reachable

http://www.codeproject.com/KB/showcase/IfOnlyWedUsedANTSProfiler.aspx
Memory leak example

- Driverless truck
  - 10,000 lines of C#
- Leak: past obstacles remained reachable
- No immediate symptoms

“This problem was pernicious because it only showed up after 40 minutes to an hour of driving around and collecting obstacles.”

http://www.codeproject.com/KB/showcase/IfOnlyWedUsedANTSProfiler.aspx
Memory leak example

• Driverless truck
  – 10,000 lines of C#
• Leak: past obstacles remained reachable
• No immediate symptoms
  “This problem was pernicious because it only showed up after 40 minutes to an hour of driving around and collecting obstacles.”
• Quick “fix”: restart after 40 minutes

http://www.codeproject.com/KB/showcase/IfOnlyWedUsedANTSProfiler.aspx
Memory leak example

• Driverless truck
  – 10,000 lines of C#
• Leak: past obstacles remained reachable
• No immediate symptoms
  “This problem was pernicious because it only showed up after 40 minutes to an hour of driving around and collecting obstacles.”
• Quick “fix”: restart after 40 minutes
• Environment sensitive
  – More obstacles in deployment
  – Unresponsive after 28 minutes

http://www.codeproject.com/KB/showcase/IfOnlyWedUsedANTSProfiler.aspx
GC basics

Two types (duals of each other):

• Reference counting
  – Work proportional to dead objects
  – Memory freed immediately
  – Cycles are problematic

• Tracing
  – Work proportional to live objects
  – Freeing postponed
  – Can be concurrent
How does tracing GC work?

Roots
• Local variables: registers & stack locations
• Static variables

Transitive closure

Memory & type safety ensure GC knows the roots and references exactly
How does tracing GC work?

When does it happen?
• Stop-the-world: safe points inserted by VM
• Concurrent
• Incremental

How many GC threads?
• Single-threaded
• Parallel
Reachability

- The runtime memory management system examines all global variables, stack variables, and live registers that could refer to objects on the heap (i.e., the **roots** of reachability).
- GC threads can **trace** these pointers through the heap (following object fields that themselves point to heap objects) to find all reachable objects.
Reachability

• Tracing collector
  – Marks the objects reachable from the roots as **live objects**, and then performs a reachability computation from them
Reachability

• Tracing collector
  – Marks the objects reachable from the roots as **live objects**, and then performs a **reachability** computation from them
Reachability

- Tracing collector
  - Marks the objects reachable from the roots as live objects, and then performs a reachability computation from them
Reachability

• Tracing collector
  – Marks the objects reachable from the roots as **live objects**, and then performs a **reachability** computation from them

• All unmarked objects are **dead**
Reachability

• Tracing collector
  – Marks the objects reachable from the roots as **live objects**, and then performs a **reachability** computation from them
• All unmarked objects are **dead**
The Big Picture

- Heap organization; basic algorithmic components

Allocation

- Free List
- Bump Allocation

Identification

- Tracing (implicit)
- Reference Counting (explicit)

Reclamation

- Sweep-to-Free
- Compact
- Evacuate
The Big Picture

- Heap organization; basic algorithmic components

**Allocation**
- Free List
- Bump Allocation

**Identification**
- Tracing (*implicit*)
- Reference Counting (*explicit*)

**Reclamation**
- Sweep-to-Free
- Compact
- Evacuate
Semispace

- Fast **bump pointer** allocation
- Requires copying collection
- Cannot incrementally reclaim memory, must free en masse
- Reserves 1/2 the heap to copy in to, in case all objects are live
Semispace

- Fast **bump pointer** allocation
- Requires copying collection
- Cannot incrementally reclaim memory, must free en masse
- Reserves 1/2 the heap to copy into, in case all objects are live

![Diagram of heap and semispace](image-url)
Semispace

- Fast **bump pointer** allocation
- Requires copying collection
- Cannot incrementally reclaim memory, must free en masse
- Reserves 1/2 the heap to copy in to, in case all objects are live

![Diagram showing semispace to space and from space](image-url)
Semispace

- Fast **bump pointer** allocation
- Requires copying collection
- Cannot incrementally reclaim memory, must free en masse
- Reserves 1/2 the heap to copy in to, in case all objects are live

![Diagram showing the Semispace concept](image-url)
Semispase

- Mark phase:
  - copies object when collector first encounters it
  - installs forwarding pointers
Semispace

• Mark phase:
  – copies object when collector first encounters it
  – installs **forwarding pointers**
  – performs transitive closure, updating pointers as it goes

```

```

from space

heap

to space
Semispace

• Mark phase:
  – copies object when collector first encounters it
  – installs *forwarding pointers*
  – performs transitive closure, updating pointers as it goes
Semispace

• Mark phase:
  – copies object when collector first encounters it
  – installs forwarding pointers
  – performs transitive closure, updating pointers as it goes
Semispace

• Mark phase:
  – copies object when collector first encounters it
  – installs forwarding pointers
  – performs transitive closure, updating pointers as it goes
  – reclaims “from space” en masse
Semispace

• Mark phase:
  – copies object when collector first encounters it
  – installs forwarding pointers
  – performs transitive closure, updating pointers as it goes
  – reclaims “from space” en masse
  – start allocating again into “to space”
Semispace

• Mark phase:
  – copies object when collector first encounters it
  – installs **forwarding pointers**
  – performs transitive closure, updating pointers as it goes
  – reclaims “from space” en masse
  – start allocating again into “to space”
Semispace

- **Notice:**
  - fast allocation
  - locality of contemporaneously allocated objects
  - locality of objects connected by pointers
  - wasted space
The Big Picture

- Heap organization; basic algorithmic components

**Allocation**
- Free List
- Bump Allocation

**Identification**
- Tracing (*implicit*)
- Reference Counting (*explicit*)

**Reclamation**
- Sweep-to-Free
- Compact
- Evacuate
Mark-and-Sweep Implementation

- Free-lists organized by size
  - blocks of same size, or
  - individual objects of same size
- Most objects are small < 128 bytes
Mark-and-Sweep Implementation

- Allocation
  - Grab a free object off the free list
Mark-and-Sweep Implementation

• **Allocation**
  – Grab a free object off the free list

```
128
16
12
8
4
```

```
heap
```

```
free lists
```
Mark-and-Sweep Implementation

- Allocation
  - Grab a free object off the free list
Mark-and-Sweep Implementation

• Allocation
  – Grab a free object off the free list
  – If there is no more memory of the right size, a garbage collection is triggered
  – Mark phase - find the live objects
  – Sweep phase - put free ones on the free list

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>...</th>
<th>128</th>
</tr>
</thead>
</table>

free lists
Mark-and-Sweep Implementation

- **Mark phase**
  - Reachability computation on the heap, marking all live objects
- **Sweep phase**
  - Sweep the memory for free objects, and populate the free lists

| 4 | 8 | 12 | 16 | ...
|---|---|----|----|---
|   |   |    |    |   |
|   |   |    |    |   |
|   |   |    |    |   |
| 128 |   |    |    |   |

free lists

heap
Mark-and-Sweep Implementation

- **Mark phase**
  - Reachability computation on the heap, marking all live objects
- **Sweep phase**
  - Sweep the memory for free objects, and populate the free lists

![Free lists diagram](image)
Mark-and-Sweep Implementation

- **Mark phase**
  - Reachability computation on the heap, marking all live objects
- **Sweep phase**
  - Sweep the memory for free objects, and populate the free lists

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>...</th>
<th>128</th>
</tr>
</thead>
</table>

free lists
Mark-and-Sweep Implementation

- **Mark phase**
  - Reachability computation on the heap, marking all live objects
- **Sweep phase**
  - Sweep the memory for free objects, and populate the free lists
Generational Collection

What objects should we put where?

• **Generational hypothesis**
  – young objects die more quickly than older ones [Lieberman & Hewitt’83, Ungar’84]
  – most pointers are from younger to older objects [Appel’89, Zorn’90]

Organize the heap in to young and old, collect young objects preferentially
Generational Heap Organization

- **Divide the heap into two spaces: young and old**
- Allocate in to the young space
- When the young space fills up,
  - collect it, copying into the old space
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space $n < m$ fills up, collect $n$ through $n-1$
Generational Heap Organization

- Divide the heap into two spaces: young and old
- **Allocate in the young space**
  - When the young space fills up,
    - collect it, copying into the old space
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space $n < m$ fills up, collect $n$ through $n-1$
Generational Heap Organization

- Divide the heap into two spaces: young and old
- Allocate in to the young space
- **When the young space fills up,**
  - collect it, copying into the old space
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space $n < m$ fills up, collect $n$ through $n-1$

![Diagram of Generational Heap Organization]

to space

Young       Old
Generational Heap Organization

- Divide the heap into two spaces: young and old
- Allocate into the young space
- **When the young space fills up,**
  - collect it, copying into the old space
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through n-1
Generational Heap Organization

- Divide the heap into two spaces: young and old
- Allocate in to the young space
- **When the young space fills up,**
  - collect it, copying into the old space
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space $n < m$ fills up, collect $n$ through $n-1$
Generational Heap Organization

- Divide the heap into two spaces: young and old
- **Allocate in to the young space**
- When the young space fills up,
  - collect it, copying into the old space
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through n-1
Generational Heap Organization

- Divide the heap into two spaces: young and old
- Allocate into the young space
- **When the young space fills up,**
  - collect it, copying into the old space
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through n-1
Generational Heap Organization

- Divide the heap in to two spaces: young and old
- Allocate in to the young space
- **When the young space fills up,**
  - collect it, copying into the old space
- When the old space fills up
  - collect both spaces
  - Generalizing to $m$ generations
    - if space $n < m$ fills up, collect $n$ through $n-1$
Generational Heap Organization

- Divide the heap into two spaces: young and old
- Allocate into the young space
- **When the young space fills up,**
  - collect it, copying into the old space
- When the old space fills up
  - collect both spaces
  - Generalizing to \( m \) generations
    - if space \( n < m \) fills up, collect \( n \) through \( n-1 \)
Generational Heap Organization

- Divide the heap into two spaces: young and old
- Allocate in to the young space
- **When the young space fills up,**
  - collect it, copying into the old space
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space $n < m$ fills up, collect $n$ through $n-1$
Generational Heap Organization

- Divide the heap into two spaces: young and old
- Allocate in to the young space
- When the young space fills up,
  - collect it, copying into the old space
- **When the old space fills up**
  - collect both spaces
  - Generalizing to m generations
    - if space $n < m$ fills up, collect $n$ through $n-1$
Generational Write Barrier

Unidirectional barrier

- record only older to younger pointers
- no need to record younger to older pointers, since we never collect the old space independently
  - most pointers are from younger to older objects [Appel’89, Zorn’90]
  - track the barrier between young objects and old spaces
Generational Write Barrier

unidirectional boundary barrier

```c
// original program
p.f = o;

// compiler support for incremental collection
if (p > barrier && o < barrier) {
    remset_nursery = remset_nursery U &p.f;
}

p.f = o;
```

to space
Young  Old
Generational Write Barrier

Unidirectional

- **record only older to younger pointers**
- **no need to record younger to older pointers, since we never collect the old space independently**
  - most pointers are from younger to older objects [Appel’89, Zorn’90]
  - most mutations are to young objects [Stefanovic et al.’99]
Garbage Collection in Real Systems

Often combine multiple kinds of collection (story)