Java-C++ Comparison

- A refresher in OO language features
- Ask questions!
Differences at a Glance

Java eliminates some of the C++ complexities

- No preprocessor (\#define, \#include, \#ifdef)
  - no biggie? (cpp -P and some shell stuff)

- No direct memory access (pointers to existing storage)

- No multiple inheritance

- No operator overloading, no implicit conversions

- No goto, free, typedef

- No unions

- No stand-alone functions, global variables
Differences at a Glance

Java adds some things to C++

- Interfaces
- Garbage collection
- Object, reflection
- Threading (and monitors) in the language
- Bound/cast checking

Java does many things differently

- Packages, inner classes
- Arrays
- Templates vs. generics
- Incremental compilation
- Virtual machine and bytecode
Interesting Changes: Packages

- Java has no global variables and functions - everything is part of a class

- Java packages are a module mechanism, like C++ namespaces
  - unlike modules in other languages, namespaces and packages are open

- Access specifiers:
  - for packages, `public` or private (by omitting public)
  - for classes, `private`, `public`, `protected`, or package-protected (by omitting all access specifiers)
References

• Java has no pointers to memory—objects and arrays are always passed “by reference-value”
  - “by reference” is another term used, but it’s overloaded (e.g., C++ reference types have
    the textbook “by reference” semantics)
    - what is the difference? How do C++ references and const references work? How are Java
      primitives passed?

• This has some side-effects:
  - copying can be done only with the clone method (on Cloneable types)
  - deep equality can be tested only with the equals method (or equivalent)
  - no direct access to memory, no pointer arithmetic, reference types treated
differently than primitive types
Garbage Collection

- Java has no explicit `free` statement, memory is reclaimed automatically

- A “finalizer” *may* be called

- What’s wrong with never reclaiming any memory?
  - address space “real estate” is cheap, isn’t it?

- Does this mean there can be no memory leaks?
  - a field is not set to `null`, expecting to be overwritten in the future
  - a local variable is not set to `null`, expecting to go out of scope soon
    - a problem in long-running methods
  - Swing/AWT listener that is not removed after it is no longer needed (either by fault of user code or by fault of system code)
  - caching strategy makes objects become unreachable more slowly
Arrays

• No support for multidimensional arrays: they are just arrays of arrays
  - this is not the same in C/C++, despite the similar syntax: an array of pointers to arrays is what’s closest to Java
    - think of the memory layout and recall that multidimensional arrays is where the array/pointer duality breaks in C

• Interesting syntax:
  - byte f[][] = new byte[128][16]
  - int i[][] = new int[100][]

• Arrays are covariant:
  if the following is legal,
    B b; A a = b;
  then the following is also legal
    A a[] = new B[];
Threads and Monitors

- Monitor-style concurrent programming:
  - using mutexes for exclusion from a critical section
  - using conditions (i.e., \texttt{wait} statements) for protected waiting
    - used for inclusion in a critical section

- Java supports concurrent programming with threads and monitors at the language level
  - \texttt{synchronized} keyword
  - “friendly” thread library
Method Overriding

• A method with the same signature can be defined and it *overrides* the superclass method - the new method is called for objects of the subclass

(what’s the difference with “overloading”?)

• Pre-Java-5, overriding method needed to have the exact same signature as the original (*non-variance*)

• In current Java (as in C++) the return type can be more specific (*covariance*)
Interfaces

• Interfaces partially describe the signature of a class
  - not sufficient for static type checking,
    unfortunately: no nested classes, no
    constructors, no final attributes, etc.

• Interface conformance needs to be explicitly declared (named, not structural conformance)

• Interfaces used to eliminate a common need for multiple inheritance
C++ Templates (10 mile-high view)

- Class templates:

```cpp
template <class E1, class E2>
struct Pair {
    E1 fst;
    E2 snd;
    ...
};
```

- Function templates:

```cpp
template <class T>
const T& max(const T& e1, const T& e2) {
    if (e1 > e2) return e1;
    else return e2;
}
```

- Templates: a full sub-language for compile-time computation
C++ Operator Overloading and Implicit Conversions

• Advanced features from a language design point of view
  - they are essentially extensibility features for C++ compilers
  - can be used to redefine the syntax and the type system of the language

• Overloading:

```c++
struct F {
  ...
  int operator[] (int index) {...}
};
```

• Implicit conversion:

```c++
class F {
  operator int() const {return 1;}
};
```