Java in a nutshell

Java: History

- 1991: Gosling et alt., for embedded systems
- 1995: Sun, first public release, Java Applets
- 1991-2004: libraries for everything
- Never standardized,
  evolves through Java Community Process
- 2006: released as Free Software
- 2010: Oracle acquires Sun
- Official programming language for Android

Java: evolution

- Java 1: mix, matchs & simplify SOME ideas taken from other languages
  - Compiled to bytecode (compile once, run (debug?) everywhere)
  - Imperative (syntax similar to C/C++)
  - Eager (code executed even if result not needed)
  - Automatic memory management (uses a Garbage Collector)
  - Strongly typed (no crashes at run-time)
  - Statically typed, but not completely (dynamic casts)
  - Extreme verbosity (no type inference)
  - Object Oriented (everything but primitive types are objects)
  - Class Based (without multiple inheritance, simplifies C++)
  - Nominal typing (an object has type T if it is an instance of a class C that implements T)
  - Parameters passed by value; objects handled by reference
  - Concurrent (based on monitors, wait for OS course)

Java: evolution

- Java 1.5: modernization, again stealing old ideas
  - Generics (aka parametric polymorphism, syntax similar to C++)
  - Annotations/metadata (for automatic code verification, automatic code generation, user provided type systems, etc.)
  - Towards iterators: enhanced for loops
  - A bit (only a bit) of type inference
  - Reflection (the program can inspect and modify itself at run-time)
- Java 1.8: towards functional programming, steals again
  - Lambda expressions/closures
  - Impure interfaces, multiple inheritance

JVM

- JVM is the Java Virtual Machine that interpretes the bytecode
- Alternatives:
  - Compilation to architecture specific native code (some language features make compilation hard/inefficient)
  - Just-in-time compilation (JIT)
    - the interpreter compiles on the fly frequently executed code after statistical analysis and dynamic linking
    - potentially faster code, but slow startup time

Languages that target the JVM

- Reimplementation of „classical“ languages
  - Jython, Jruby, Javascript, …
- New languages, mostly functional, better than Java
  - Scala, Groovy, Clojure
  - Example Scala vs Java:
    - Type inference (less verbose code)
    - Functional (better distinction mutable vs immutable, immutable data structures in the standard library, functions as first class objects, no distinction expression/command)
    - Lazy evaluation on demand (execute code only if needed)
    - Abstract Data types (aka case classes)
    - Structural typing (an object has a type if it can do what the type requires, not if it was declared to have that type
    - …
- Can reuse for free Java libraries

Java in a nutshell

- Google for „Java Tutorials“ to learn the language
- Only a bit of the language used in this course

HelloWorld

```
/**
 * The HelloWorldApp class implements an application that
 * simply prints "Hello World!" to standard output.
 */
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Display the string.
    }
}
```

javac HelloWorld.java To compile; yields HelloWorld.class (bytecode)

java HelloWorld To run the public static method main

static vars and „methods“

A class plays two distinct roles in Java (very bad!)

1. It restricts visibility of „global“ declarations
   - Like modules/blocks in other programming languages (not in C)
2. It is a blueprint for objects that specifies their fields and methods
   - In Java (and C++) a class is both a blueprint for an object and a declaration of a type of objects (bad!)

Parameter passing convention

Java seems to have a non uniform parameter passing convention (until you understand what a reference is!)

- **Primitive types** (int, float, boolean, …) are passed **by value**
  - A copy of the value is passed
  - Changes to the copy do not affect the original ofc
- The **references** to arrays and objects are passed **by value**
  - T[] x declares a reference (a pointer, but don't tell anyone...) to an array
  - classname x declares a reference (again a pointer) to an object of class classname
  - References are passed by copying them (as for primitive types), but it is just a copy of a pointer
  - **The two references point to the same data in memory: changes are visible after the call**
- All local references MUST be initialized (no dandling pointers, very good!) but global ones are initialized to null and you can always assign null to a reference (bad!)
- No pointer arithmetics (very very good!):
  - you can't take the address of something (no &)
  - you can't dereference a reference/pointer (no *)
  - you can't forge an address(not int* p = 3453525;)

See ParameterPassing.java

Java is imperative

- Syntax borrowed from C:
  - Assignment: x=4;
  - If-then-else: if(guard) cmd1 else cmd2; if(guard) cmd1;
  - Blocks: { … }
  - Switch/break/default on numbers, strings, enumerated types, …
  - while(guard) body;   do body while(guard);
  - for(initialization; guard; increment) body
    unlike C, you can declare variable in initialization
    e.g. for(int i=0; i<10; i++) { … }
  - Enchanced for: for(type var : collection) body

    int [] a = {0,1,2};
    for (int n : a) System.out.printnl(n); // prints 0 1 2

    Works on arrays and all classes that implement the Collection interface
  - break, continue
  - return

Java uses eager evaluation

- When calling f(E), E is immediately evaluated BEFORE calling f
  - The side effects of E comes before and only once
  - E evaluated even if f won't use it
- Example:

```
static int g() {
  System.out.println(„Inside g“));
  return 0;
}
…. f(g()) …
```

See EagerEvaluation.java

Classes, inheritance and subtyping

See Classes.java for:

- How to define a class
  - Fields, methods, constructors
- How to inherit from a superclass
  - Method overriding, calling the constructors of the superclass
- Late binding: the code to be invoked is determined AT RUN TIME looking at the class the object instantiated at creation time
- How to create objects
- Subtyping
- Garbage collection

Interfaces, overloading, dynamic casts

See Interfaces.java for:

- How to define an interface
- How to declare constants
- How to overload a methods
- Multiple inheritance for interfaces
- Dynamic casts (bad)

Interfaces, abstract classes

See Interfaces2.java for:

- Default and static methods in interfaces (and the related multiple inheritance mess)
- Abstract classes
  - Classes with abstract instance methods = methods declared but not implemented yet
  - Cannot be instantiated
  - Useful to inherit shared code or to partially implement interfaces
- Why both abstract classes and interfaces? (rather arbitrary choices, bad)

More on classes

- Poor man ways to prevent subclasses from destroying the super-class invariants:
  - Declare the class as final to avoid subclassing
    - But no safe extensions either
  - Declare an instance method as final to avoid overridding in a subclass
- Objects must be constructed in a well-disciplined way:
  - Other constructors must be invoked at beginning of a constructor
  - Only final/static methods should be called in a constructor

Inner and Static Nested Classes

See NestedClasses.java for:

- Declaring inner classes and static inner classes (called static nested classes)
  - To localize code close to its use
  - To access the state without making everything public
  - To avoid pollution of the APIs (if the inner classes are private)
- Special syntax to instantiate objects of an inner class or a static nested class
- Special syntax to refer to names shadowed in the outer class

Uniformity

- Principle of uniformity:
  - Primitive types and all references to complex types have exactly the same size (i.e. the size of a pointer)
  - Therefore the code to write/read/store a value is exactly the same independently from the type of the value
- Example: these two functions emit the same code

```
void swap(String[] a)
   { String t; t = a[0]; a[0] = a[1]; a[1] = t }

void swap(int[] a)
   {  int t;       t = a[0]; a[0] = a[1]; a[1] = t }
```

Generics

- Relaxed principle of uniformity (e.g. C++):
  - Different types may have different size
  - The code to write/read/store a value is only parameterized by the size of the type
- Example: the code for the two previous swap functions is exactly the same, up to the constant used for pointer arithmetics

Parametric polymorphism

- Parametric polymorphism (assumes the uniformity principle):
  - The user writes once and for all the code of a function that works uniformly on values
  - The compiler emits the code once & forall
  - The user calls the code in multiple places on different types; no work needs to be done either at compile or run time
- Example (in OCaml):

```
(* val swap: ∀a,b.  a * b → b * a *)
let swap (x,y) = (y,x)
swap(1,2); swap("foo","goo")
```

Typing for parametric polymorphism

- How to type a function that works on ALL types? Using an universal quantifer!
  - Example:
    - Sort: ∀a. a[ ] → a[ ]
      sorts an array of every type
      by the forall elimination rule

$$\frac{\forall a.\ a[\,] \rightarrow a[\,]}{\text{String}[\,] \rightarrow \text{String}[\,]}$$

thus I can sort an array of String (or int, or whatever)

Generics

- Introduced first in C++ (then Java, then …)
- A very heavy syntax to declare classes/functions/methods/etc. that are universally quantified over one or more TYPE variables (i.e. classes or interfaces in Java)
  - Example: static <E> E returnNth (E[] v, int n);
    means ∀E. E[] → int → E
- A very heavy syntax to instantiate the quantifier to a concrete type
  - Example: Class.<String> returnNth(argv,0);
    means apply ∀-elimination using String
- The compiler may emit just one code (if uniformity holds), or several

Generics in Java

- In Java: principle of uniformity limited to references
  - Example:
    - Util.<String> swap(s1,s2); // Ok, s1 and s2 // references
    - Util.<int> swap(4,5); // Not ok, int is a // primitive type
- Partial (bad) type inference (good):
  - Sometimes (??) the type instantiation can be omitted when the compiler is able to infer the type
  - Example: swap(s1,s2) in place of Util.<String> swap(s1,s2)

Generics in Java

See generics.java for:

- Generic classes
- Generic interfaces
- Generic methods

Not in this course (very very very bad):

- Raw types, i.e. you define, say, a Pair<T,U>
  but you instantiate it as a Pair holdings two objects of UNKNOWN type
- All checks no longer performed at compile time
- Exceptions raised at run time in case of type mismatch

Bounded polymorphism

See BoundedGenerics.java for:

- Parameters <E extends T>
  to assume E to be a sub-type of T
  (and be able to invoke methods of T)

- General case:
  <E extends C & I1 & … In>
  where
  - C is an optional class
  - I1, …, In are interfaces
  - Note: you cannot use two classes as bounds
    (why, oh , why?)

Bounded Polymorphism

See BiBoundedGenerics.ma for:

- How to bound a type parameter from below
- How to use double bounding (from above and from below) to allow containers for comparable objects without losing subtyping

Exceptions

- Events used to signal exceptional behaviour of the code

1) Errors that can be anticipated and recovered: e.g.
   - Opening a file that does not exists
   - Reading a wrong password
   - The input to a function is invalid
2) Errors that cannot be anticipated or recovered: e.g.
   - Memory exhausted
   - Hardware errors
3) Violations of the program logic: e.g.
   - A list should be non empty, but it is not
   - An element found twice in a list without repetitions

Exceptions

An exceptions e can be:

- Thrown with: **throw**(e);
- Handled with:
  try { … *exceptions are raise here* … }
  catch (ExceptionType1 name) { … }
  …
  catch (ExceptionTypen name) { … }
  finally { … }
- Raising interrupts normal program execution; control goes to the innermost enclosing catch/finally block.
- If there is no catch block, the exceptions is recursively propagated to the caller.
- The finally block is executed anyway even if no exception is raised

Exceptions

- An exception is an instance of a class:
  - Errors inherit from the **Error** class
  - Control flow violations inherit from **RuntimeException**
  - All other exceptions are called **checked exceptions** and inherit from **Exception**
  - You can define your own and use them to transfer data from the raising to the catching point
    try { … if (x<y) raise (new BadData(x,y)); … }
    catch (BadData e) { printnl(e.GetMessage()); }

Exceptions

If a method does not catch a checked exception, it must declare that the exception can escape it.

```
class MyException extends Exception;

int MyMethod(int x) throws MyException {
    if (x<0)
        throw new MyException(„Negative index“);
}
```