

600.226 Java intro

Subodh Kumar
Johns Hopkins University

Sun's Java Tools

- javac : Java byte code compiler
 - ◆ Compiles Java source to platform-independent byte codes
- java : Java run-time environment
 - ◆ Verifies byte codes for security correctness and executes on Java Virtual Machine
- jdb : Java debugger
- forte: Graphical debugging environment

Available free from <http://java.sun.com>

Java: A high level objected oriented programming language

- Object Oriented language; like C++
 - ◆ Class, Object, method
 - ◆ No */& address ops
 - ◆ No free/delete/malloc
- Architecture independent
- In-built:
 - ◆ Graphics
 - ◆ Networking
 - ◆ Thread management
 - ◆ Garbage collection
 - ◆ Error/Exception handling Handling

Some Shortcomings

- Rather slow compared to fully-compiled code
 - ◆ Changing with on-the-fly compilation technology
- Some unavoidable space inefficiencies
- Difficult to take advantage of platform-specific features

Polymorphism

- Ability of variable to take on many forms
 - ◆ Class variable may contain exact class or any descendent
 - ◆ Interface variable may contain any class implementing the interface
- Allows for greater modularity

Using Inheritance

- Specialization
 - ◆ Override some parent methods
 - ☞ Refinement: call parent method and then do more
 - ☞ Replacement: just do something different
 - ◆ Handle differences in behavior between parent and child
- Extension
 - ◆ Add to the functionality of parent by adding new data and behaviors
- (Real examples often do some of both)

Primitive Types	
boolean	true or false
char	16-bit Unicode character
byte	8-bit signed integer
short	16-bit signed integer
int	32-bit signed integer
long	64-bit signed integer
float	32-bit floating point
double	64-bit floating point

- | Differences from C++ | |
|--|--|
| <ul style="list-style-type: none">■ boolean true and false do not have integer equivalents■ char is not a byte, but a 16-bit Unicode■ No unsigned integer types (so byte goes to 128, not 256)■ All types have specified size | |

Classes

- Combine *fields* (variables) and *methods* (procedures)
- Fields and methods accessed by . (dot) operator

```
class MyClass {
    static int numInstances=0;
    protected int somethingImportant;
    public int tellAll() {
        return somethingImportant;
    }
}
```

```
MyClass myVar = new MyClass;
System.out.println(MyClass.numInstances + ", " +
    myVar.tellAll());
```

Declaring Classes

```
public class Gnome {
    // body
}
```

**Extended/instantiated within package / by importer
otherwise only within the package**

```
abstract class Gnome { }
```

Only abstract methods

```
final class Gnome { }
```

No subclasses allowed

Local variables

```
class Gnome {  
  // Instance Variables  
  public Tot[] tots;           // class Tot is visible here.  
                               // tot1 is exported with Gnome  
  protected String name;     // Visible to package and  
                               // sub-classes  
  private int age;           // Visible Only within the class  
  static final int NTOT = 2; // Visible within package  
                               // Value can't be changed  
  
  // Methods  
}
```

Methods

```
abstract class Gnome {  
  // Instance Variables  
  
  // Methods  
  public abstract void hiFunda(int party);  
  final int getAge() {  
    return age;  
  }  
  // May be public, protected, private  
  // static, final, and abstract (if the class is abstract)  
}
```

Fields

Modifiers

- ◆ static : affects instantiation
- ◆ public, protected, private : affect visibility
- ◆ final : makes field a constant
- ◆ synchronized : used for multithreading

Field Instantiation

- Class fields
 - ◆ static - only one per class
 - ◆ May be accessed without a class variable
 - ☞ `<classname>.<static field>`
 - e.g. `Math.PI`
- Instance fields
 - ◆ non-static - one per class instance

Field and Method Visibility

public, protected, private, or “package” (default)

Accessible to:	public	protected	package	private
same class	yes	yes	yes	yes
class in same package	yes	yes	yes	no
subclass in different package	yes	yes	no	no
non-subclass, different package	yes	no	no	no

- Methods
- **Modifiers**
 - ◆ Same as fields, plus abstract
 - **May be overloaded (methods with same name)**
 - ◆ Must have different signatures (defined by parameter type sequence)
 - **static** methods cannot access instance variables
 - **this** provides reference to class instance
 - ◆ Can be passed as parameter to another method
 - ◆ Can disambiguate class fields from parameters

Instance initialization: Constructors

```
class Gnome {  
  // Constructor Method  
  public Gnome (String x) {  
    name = x;  
    tots = new Tots[NTOT];  
  }  
  // May be public, protected, private  
  // static, final, abstract not allowed  
}  
Usage: Gnome gnome1 = new Gnome("Baital")
```

Initialization/Constructors

Initialization performed when class is *instantiated* by:

```
new <class>[ (params) ]
```

- ◆ Fields initialized to specified values or to defaults according to type
- ◆ Constructor called if there is one (params must match a constructor signature)
- ◆ Constructors may be overloaded as well

Inheritance

Enables class extensions with reuse of fields and methods

- ◆ All parent fields included in child instantiation
- ◆ Protected and public fields and methods directly accessible to child
- ◆ Parent methods may be *overridden*
- ◆ New fields and methods may be added to child
- ◆ Only single inheritance (unlike C++)

Deriving sub-classes

```
class Gnome extends Dreblin {  
    // body  
}  
    Redefine/Add to variables/methods.  
class Gnome implements Dreblin, Timken {  
    // body  
}  
    Define methods of an interface
```

Initialization of Derived Classes

- `super()`: alias for constructor of parent class
 - ◆ Constructor of derived class can explicitly call `super()` (with or without arguments) to invoke parent constructor
 - ◆ If constructor does not call `super()` or `this()` at start of constructor
 - ☞ `super()` is automatically called (with no arguments)
 - ◆ Inside `super()` function, this object has been cast to parent class

Simple Inheritance Example

```
Class MyExtension extends MyClass {  
    float newField;  
    MyExtension() {  
        super(); //call parent constructor  
        newField = super.tellAll()*3.14;}  
    int tellAll() {return (int)newField;}  
}
```

```
MyClass foo = new MyExtension;  
System.out.println(foo.tellAll());
```

- Method accessed is that of actual instantiation type, not variable type
 - ◆ In C++ terminology, all functions are “virtual”

Casting of Class Variables

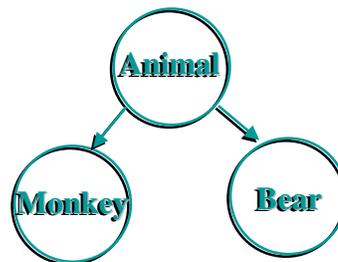
- “upward” casting
 - ◆ Casting derived class variable to ancestor class is always safe (and may be done implicitly)
- “downward” casting
 - ◆ Casting class variable to derived class fails if variable is not actually an instance of the derived class
 - ☞ run-time error
 - ◆ **instanceof ()** function can be used to test class type before downward cast

Class Casting Example

```
class Animal {...}
class Bear extends Animal {...}
class Monkey extends Animal {...}
```

```
Animal a;
Monkey m;
Bear b;
```

```
a = new Bear();
b = (Bear) a;
m = (Monkey) b;
m = (Monkey) a;
```



Class Casting Example

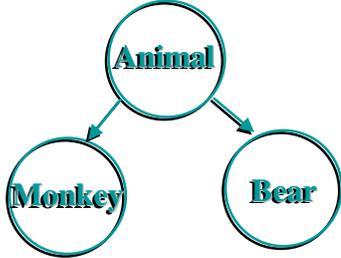
```

class Animal {...}
class Bear extends Animal {...}
class Monkey extends Animal {...}

Animal a;
Monkey m;
Bear b;

a = new Bear();           // legal
b = (Bear) a;             // legal
m = (Monkey) b;          // illegal
m = (Monkey) a;          // illegal

```



```

graph TD
    Animal((Animal))
    Monkey((Monkey))
    Bear((Bear))
    Monkey --> Animal
    Bear --> Animal

```

Abstract Classes

- Include one or more unimplemented abstract methods
- Enable inheritance of methods that don't make sense at the parent level

```

abstract class Shape {
    abstract public draw(); }
class Circle extends Shape {
    public draw() {...} }
class Rectangle extends Shape {
    public draw() {...} }
Shape s = new Circle();
s.draw();

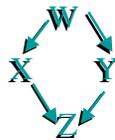
```

Interfaces

- Similar to abstract classes
 - ◆ But *no* methods implemented or fields specified
- Class can be defined to *implement* one or more interfaces
 - ◆ More general mechanism than just single inheritance

```
interface Printable {  
    print(); }  
class Foo implements Printable {  
    print(){...}; }
```

"Multiple Inheritance"



- Several ways to achieve in Java
 - ◆ combinations of interfaces and classes
 - ◆ W and Y are interfaces, X and Z are classes
 - ◆ W, X, and Y are interfaces, Z is class

Exceptions

- Language-level support for managing run-time errors
- You can define your own exception classes
- Methods declare which exceptions they might possibly *throw*
- Calling methods either *catch* these exceptions or pass them up the call stack

Throw

```
public void myMethod() throws BadThingHappened {  
    ...  
    if (someCondition)  
        throw new BadThingHappened;  
    ...  
}  
  
try  
    ...  
    myMethod()  
    ...  
catch (BadThingHappened BTH)  
    { block of code }  
catch (exceptiontype id)  
    { block of code }  
finally  
    { block of code } // ALWAYS executed!!
```