

Enumerations, Generics, and Packages

Lecture 3

Constant fields: final

- Modifier *final* on field means it cannot change
 - For primitive type, effectively a constant

```
final int i1 = 53;
final int i2 = (int) (Math.random()*20);
final int i3;
. . .
i2++;
```

 ← Compile-time Error
 - For objects, only the *reference* is constant

```
final Pencil p = new Pencil("blue");
. . .
p = new Pencil();
p.sharpen(3);
```

 ← Compile-time Error
- Can be used in conjunction with static
 - Class-wide constant value

```
static final DEFAULT_LENGTH = 14;
```

Outdated (bad) Idiom: int enums

- Enumeration type: legal values a finite set of constants
 - Card suits (clubs, diamonds, hearts, spades)
 - Days of the week (D, M, T, W, R, F, S)
- This could be done with static final fields

```
class PlayingCard {
    public static final int CLUBS = 0;
    public static final int DIAMONDS = 1;
    public static final int HEARTS = 2;
    public static final int SPADES = 3;
    . . .
}
```
- Later, use these named constants

```
int trump = . . . ;
if (trump == PlayingCard.CLUBS) { . . . }
```
- Problem: no type safety! *trump* is just an int

```
if (trump == 23) { . . . }
```

Enum Types

- Declared like a class, keyword *enum*
 - Contains a list of *enum constants*

```
enum Suit {
    CLUBS, DIAMONDS, HEARTS, SPADES
}
```
 - These constants are (implicitly) static fields

```
Suit trump = Suit.SPADES; //do not use new()
if (trump == Suit.CLUBS) { . . . }
```
- Can also contain fields & methods (and nested types)
- Automatically provided methods include:
 - `values()` – returns array of constants

```
Suit.values()[0] == Suit.CLUBS;
```
 - `valueOf(String)` – returns constant with that name

```
Suit.valueOf("CLUBS") == Suit.CLUBS;
```
 - `ordinal()` – returns constant's position in declaration list

```
Suit.CLUBS.ordinal() == 0;
```

Initialization Block

- Statement block outside methods/constructors
- Executed *before* the body of any constructor

```
Without initialization block
class Body {
    private long idNum;
    private String name = "";
    private Body orbits;
    private static long nextID = 0;

    Body() {
        idNum = nextID++;
    }

    Body(String name, Body orbits)
    {
        this();
        this.name = name;
        this.orbits = orbits;
    }
}
```

```
With initialization block
class Body {
    private long idNum;
    private String name = "";
    private Body orbits;
    private static long nextID = 0;

    {
        idNum = nextID++;
    }

    Body(String name, Body orbits)
    {
        this.name = name;
        this.orbits = orbits;
    }
}
```

Static Initialization Block

- Similar to initialization block, but:
 - Can only reference static members
 - Executed only once, when class is first loaded
- ```
class Primes {
 static int[] primes = new int[4];

 static {
 primes[0] = 2;
 for(int i = 1; i < primes.length; i++) {
 primes[i] = nextPrime(i);
 }
 }
 //declaration of static nextPrime(int) . . .
}
```

## Packages

- A *package* is a grouping of classes
  - Hierarchical: subpackages within packages
  - Sun standard libraries organized in packages
    - java.lang, java.util, java.util.logging
    - see <http://java.sun.com/j2se/1.5.0/docs/api>
- A package provides
  - Logical structuring: related classes are bundled
  - Encapsulation: another level of access control
  - Distinct namespace: classes in different packages can have the same name without conflict
    - *Convention* to guarantee uniqueness of package name: reverse of company's domain name
    - org.w3c.dom, edu.ohio-state.cse

## Declaration

- Use package statement at top of source file
  - Must appear first, before any class declarations

```
package edu.ohio-state.cse;
. . .
class Pencil { . . . }
```
- Put this file in a directory matching package name
  - Pencil.java in ???/edu/ohio-state/cse
- At most one package declaration in a file
- If there is no package declaration, class is in unnamed default package
  - This is fine for very small programs (like the ones you will write for this class)

## Access Control

- Another level of visibility: package
  - Default for members (public/private omitted)
  - Package-visible members are accessible by all classes in the same package

```
package edu.ohio-state.edu;
class Pencil {
 private String color;
 int length;
 . . .
}
```
- Classes are public or package (default)
  - Public classes available outside package

```
public class Math { . . . }
```
  - Package classes available only within same package

```
class Pencil { . . . }
```

## Type Imports

- Fully-qualified type name is *package.class*

```
java.util.Date d = new java.util.Date();
```

  - Do not confuse this "." with member access
- Shorthand: import statement at top of file
  - To import a single *public* type

```
import java.util.Date;
Date d = new Date();
```
  - To import all *public* types, use wildcard \*

```
import java.util.*;
Date d = new Date();
```

    - \* does not import subpackages
- All classes implicitly import java.lang.\*
- Static members can be explicitly imported

```
import static java.lang.Math.exp;
exp(x); //instead of Math.exp(x)
```

  - Can use wildcard \* as well

## Best Practices: Naming Conventions

- Avoid name conflicts with packages and reserved keywords
- Package names: lowercase letters
  - java.util, java.net, java.io, . . .
- Class names: start with uppercase letter
  - Math, Pencil, PriorityQueue, . . .
- Variable, field and method names: start with lowercase letters
  - x, out, myColor, abs(), getName(), isEven() . . .
- Constant names: all uppercase letters
  - PI, DEFAULT\_LENGTH, DAY\_OF\_WEEK . . .
- Type parameters: single letter upper case
  - E (element) T (type) V (value type)

## Genericity: Background

- Methods are parameterized by the *values* of their formal arguments

```
void enableLaunch (boolean go) { ... }
```

  - In a sense, there are 2 enableLaunch()'s:
    - one where go begins with value true
    - one where go begins with value false
  - Could define enableLaunchT(), enableLaunchF()

```
boolean isEven (int i) { ... }
```
  - In a sense, there are 4,294,967,296 versions of isEven() (half return true, half return false)
    - Could (?) define isEven0(), isEven1(), isEven2(), ...

```
void println (String s) { ... }
```
    - In a sense, there are ?? versions of println()
- Classes can be parameterized by the *types* of their "formal type arguments"

## Generics: Motivation

- Consider a box that holds a pencil
  - See BoxOfPencil.java
- Now consider a box that holds a string
  - See BoxOfString.java
- These two class definitions differ *only* in:
  - The type of the private field value
  - The argument type of the 1-arg constructor
  - The return type of remove()
  - The argument type of insert()
- All the rest of the code is identical!
- BoxOfPencil and BoxOfString are like two instantiations of a generic class definition
  - Parameterized by *type* (not value)

## Example: Generic Box

- Declaration

```
class Box<E> { . . . }
```
- In body of class declaration, E can now be used as a type

```
private E value;
public void insert(E value) { . . . }
```
- See Box.java
- To use generic type: classname<type>

```
Box<String> bs = new Box<String>("hi");
Box<Pencil> bp = new Box<Pencil>();
String s = bs.remove();
bp.insert(new Pencil());
Pencil p = bs.remove();
```

Compile-time Error

## Type Erasure

- Note: Box<Pencil> and Box<String> are *not* two separate classes
  - They are two generic type *invocations* of one class, Box

```
Box<Pencil> b1 = new Box<Pencil>();
Box<String> b2 = new Box<String>();
b1.getClass() == b2.getClass();
```
- Think of <Pencil> as constructor information, so the compiler can do appropriate casting and type checking
- At run-time, no generic type information remains in Box objects
  - The type parameter, E, has been "*erased*"
  - Left with one class: Box<?>

## Consequences of Type Erasure

- All type-instances share the same static members

```
static int nextID; //shared by all Box<*>
```
- Static members can not refer to naked type

```
private static E value; //compile error
```
- New instances and arrays of naked type can not be created

```
E value = new E(); //compile error
E[] myArray = new E[50]; //compile error
```
- Casts ignore parameter type information

```
Box<String> x = (Box<String>) b; //unchecked
Box<?> y = (Box<?>) b; //ok
```