

# Java Review

Reading (Ch1 from Weiss)

# Overview

- Constructors
  - Called to create new instances of a class
  - Default constructor initializes all fields to default values (0, null, false)

```
class Thing {  
    int val;  
  
    Thing(int val) {  
        this.val = val;  
    }  
  
    Thing() {  
        this(3);  
    }  
}
```

```
Thing one = new Thing(1);  
Thing two = new Thing(2);  
Thing three = new Thing();
```

# Initializing Fields

- you can provide an initial value for a field in its declaration:

```
public class BedAndBreakfast {  
    // initialize to 10  
    public static int capacity = 10; //  
    / initialize to false  
    private boolean full = false;  
}
```

- not necessary to declare fields at the beginning of the class definition
  - but this is the most common practice.
- only necessary that they be declared and initialized before they are used.
- initialization at the point of declaration works well when
  - the initialization value is available and
  - the initialization can be put on one line.
- this form of initialization has limitations?

# Initializing Fields

- has limitations because of its simplicity
- if initialization requires some logic
  - for example, error handling or
  - a for loop to fill a complex array,
- simple assignment is inadequate.
- Instance variables can be initialized in constructors, where error handling or other logic can be used.
- To provide the same capability for class variables, the Java programming language includes ***static initialization blocks***.

# Static Initialization

- *static initialization block* → normal block of code enclosed in braces, { }, and preceded by the static keyword.
- example:  
**static { // whatever code is needed for initialization goes here }**
- A class can have any number of static initialization blocks, and they can appear anywhere in the class body.
- static initialization blocks are called in the order that they appear in the source code.
- **What Is It?**
  - A block of code '{ }' that **runs only one time**, and it is **run the first time the Constructor or the main() method for that class is called → if only a reference is created before initialization, it is not called.**
- **How Does It Work?**
  - Just declare a static { } block inside your Java class and throw some code in it
  - No return statement
  - No access to this or super
  - Can throw Unchecked Exceptions ( RuntimeException, Error, and their subclasses) only
- **What Can It Be Used For?**
  - Loading drivers and other items into the namespace.
  - Initialize your complex static members once (possibly for singletons/etc)
  - ..

# Static Initialization-Example

```
public class test {  
    private static int sqr[];  
    private static final int N = 100;  
    static {  
        sqr = new int[N];  
        for (int i = 0; i < N; i++)  
            sqr[i] = i * i;  
    }  
    public static void main(String args[]) {  
        System.out.println("17 squared is " + sqr[17]);  
    }  
}
```

```
class Loader {  
    static final String theName = "The Loader";  
    static {  
        System.out.println("Loader.static");  
    }  
    Loader() {  
        System.out.println("Loader.Loader()");  
    }  
}
```

```
class Test {  
    static {  
        System.out.println( "Test.static");  
    }  
    Test() {  
        System.out.println( "Test.Test()");  
        Loader l;  
        System.out.println( Loader.theName );  
    }  
    public static void main( String [] args ) {  
        System.out.println( "Test.main");  
        Test t = new Test();  
        System.exit(0);  
    }  
}
```

# What is the output?

```
class Loader {  
    static final String theName = "The Loader";  
    static {  
        System.out.println("Loader.static");  
    }  
    Loader() {  
        System.out.println("Loader.Loader()");  
    }  
}
```

```
class Test {  
    static {  
        System.out.println( "Test.static");  
    }  
    Test() {  
        System.out.println( "Test.Test()");  
        Loader l = new Loader();  
        System.out.println( Loader.theName );  
    }  
    public static void main( String [] args ) {  
        System.out.println( "Test.main");  
        Test t = new Test();  
        System.exit(0);  
    }  
}
```

# What is the output?



# Example-Static vs Instance Variable

```
class Widget {  
    static int nextSerialNumber = 10000;  
    int serialNumber;  
  
    Widget() {  
        serialNumber = nextSerialNumber++;  
    }  
  
    public static void main(String[] args) {  
        Widget a = new Widget();  
        Widget b = new Widget();  
        Widget c = new Widget();  
        System.out.println(a.serialNumber);  
        System.out.println(b.serialNumber);  
        System.out.println(c.serialNumber);  
    }  
}
```

# A Common Error

```
class Thing {  
    int val;  
  
    boolean setVal(int v) {  
        int val = v;  
    }  
}
```

- local variable shadows field
- you would like to set the instance field **val = v**
- but you have declared a new local variable **val**
- assignment has no effect on the field **val**

# The main Method

Can be called from anywhere

A class method; don't need an object to call it

No return value

Method must be named `main`

```
public static void main(String[] args) {  
    ...  
}
```

Parameters passed to program on command line

# Names

- **this: used by an object to refer to itself**
- Refer to static and instance fields & methods of **this** by (unqualified) name:
  - **serialNumber, nextSerialNumber**
- Refer to static fields & methods in another class using name of the class
  - **Widget.nextSerialNumber**
- Refer to instance fields & methods of another object using name of the object
  - **a.serialNumber**
- Example: **System.out.println(a.serialNumber)**
  - **out** is a static field in class **System**
  - The value of **System.out** is an instance of a class that has an instance method **println(int)**

# Overloading of Methods

- A class can have several methods of the same name
- But all methods must have different *signatures*
- The *signature of a method is its name plus types of its parameters*
- Example: **String.valueOf(...)** in Java API
- There are 9 of them:
  - **valueOf(boolean);**
  - **valueOf(int);**
  - **valueOf(long);**
  - ...
- Parameter types are part of the method's signature

# Primitive vs Reference Types

- **Primitive types**

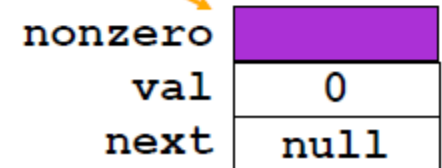
- **int, short, long, float, double, byte, char, boolean**
- efficient
- At most 64 bits
- not an **Object**—*unboxed*
- Java has primitive wrapper classes corresponding to each primitive type: Integer and **int**, Character and **char**, Float and **float**, etc.
- Boxing(wrapping)→process of placing a primitive type within an object so that the primitive can be used as a reference object.
- For example, lists work with objects, so wrapping is needed

x: 

true
------

x: 

--

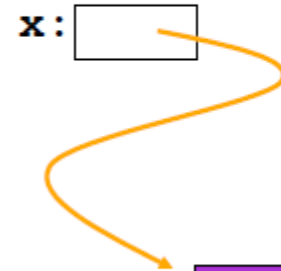



# Primitive vs Reference Types

- Reference types

- objects and arrays
- **String**, **int[]**, **HashSet**
- usually require more memory
- can have special value **null**
- can compare **null** using **==**, **!=**
- generates **NullPointerException** if you try to dereference **null**

x: true



nonzero	
val	0
next	null

# == vs equals( )

- == is an operator
- equals is a method defined in the Object class
- == checks
  - if two objects have the same address in the memory
  - for primitive it checks if they have the same value.
- equals method checks if the two objects which are being compared have an equal value (depending on how the equals method has been implemented for the objects)
- To compare object *contents*, override ***Object.equals()***
  - **boolean equals(Object x);**



# == and equals() for String

- Since Strings are objects, the equals method will return true if two Strings have the same contents
  - i.e., the same characters in the same order.
- The == operator will only be true if two String references point to the same underlying String object.
  - two Strings representing the same content will be equal when tested by the equals(Object) method
  - but will only be equal when tested with the == operator if they are actually the same object.
- To save memory (and speed up testing for equality), Java supports “**interning**” of Strings.
- When the intern() method is invoked on a String, a lookup is performed on a table of interned Strings.
- If a String object with the same content is already in the table, a reference to the String in the table is returned.
- Otherwise, the String is added to the table and a reference to it is returned.

# `==` and `equals()` for String

- The result is that after interning, all Strings with the same content will point to the same object.
- This saves space, and also allows the Strings to be compared using the `==` operator, which is much faster than comparison with the `equals(Object)` method.

## `==` VS `equals()`

`"xy" == "xy"`

`"xy".equals("xy")`

`"xy" == "x" + "y"`

`"xy".equals("x" + "y")`

`"xy" == new String("xy")`

`"xy".equals(new String("xy"))`

## == VS equals ( )

```
"xy" == "xy"
```

```
true
```

```
"xy".equals("xy")
```

```
true
```

```
"xy" == "x" + "y"
```

```
true
```

```
"xy".equals("x" + "y")
```

```
true
```

```
"xy" == new String("xy")
```

```
false
```

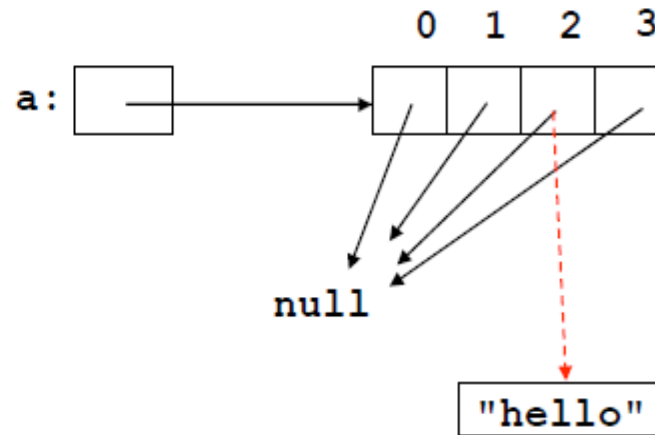
```
"xy".equals(new String("xy"))
```

```
true
```

# Arrays

- Arrays are reference types
- Array *elements* can be reference types or primitive types
  - E.g., `int[]` or `String[]`
- If `a` is an array, `a.length` is its length
- Its elements are `a[0]`, `a[1]`, ..., `a[a.length-1]`
- The length is fixed

```
String[] a = new String[4];  
a[2] = "hello"
```



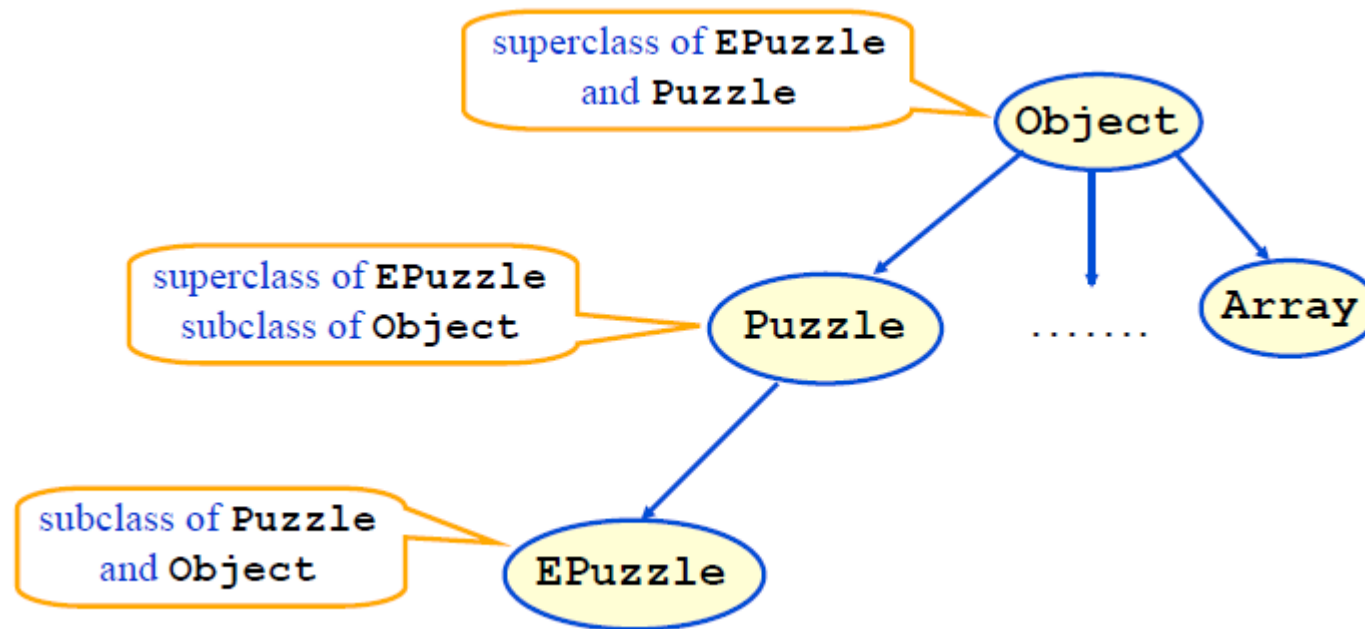
```
a.length = 4
```

## Accessing Array Elements Sequentially

```
public class CommandLineArgs {  
  
    public static void main(String[] args) {  
  
        System.out.println(args.length);  
  
        // old-style  
        for (int i = 0; i < args.length; i++) {  
            System.out.println(args[i]);  
        }  
  
        // new style  
        for (String s : args) {  
            System.out.println(s);  
        }  
    }  
}
```

# Class Hierarchy

- Every class (except **Object**) has a **unique** immediate superclass, called its *parent*



# Overriding

- A method in a subclass overrides a method in superclass if:
  - both methods have the same name,
  - both methods have the same signature (number and type of parameters and return type), and
  - both are static methods or both are instance methods.
- Methods are dispatched according to the runtime type of the object



# Accessing Overridden Methods

- Suppose a class **S** overrides a method **m** in its parent
  - Methods in **S** can invoke the overridden method in the parent as **super.m()**
  - cannot compose super more than once as in **super.super.m()**
- An overriding method cannot have more restricted access than the method it overrides

```
class A {  
    public int m() {...}  
}  
  
class B extends A {  
    private int m() {...} //illegal!  
}  
  
A supR = new B(); //upcasting  
supR.m(); //would invoke private method in  
class B at runtime!
```

# Shadowing

- Like overriding, but for fields instead of methods
  - Superclass: variable **v of some type**
  - Subclass: variable **v perhaps *of some other type***
  - Method in subclass can access shadowed variable using **super.v**
- Variable references are resolved using *static binding* (i.e., at compile-time), not *dynamic binding* (i.e., not at runtime)
  - Variable reference **r.v** uses the ***static type (declared type) of the variable r, not the runtime type of the object referred to by r***
- Shadowing variables is bad programming practice and should be avoided

# Example

```
public class Test {  
    public static void main(String[] args)  
    {  
        f(new ST2());  
    }  
    static void f(ST1 x) {  
        System.out.println(x.a);  
        System.out.println(x.b());  
    }  
}  
class ST1 {  
    String a = "ST1";  
    String b() {  
        return "ST1";  
    }  
}  
class ST2 extends ST1 {  
    String a = "ST2";  
    String b() {  
        return "ST2";  
    }  
}
```

What is the output?

\* This program demonstrates that instance method names are resolved at runtime using the runtime (dynamic) type of the object, whereas instance field names are resolved at compile time using the compile-time (static) type of the expression.

\* Inside the method f, the compile-time type of x is ST1, and this is the information used to access x.a; whereas when f is called with an object of type ST2, this is the information used to determine which b() method to dispatch.

# Java interface

```
interface IPuzzle {  
    void scramble();  
    int tile(int r, int c);  
    boolean move(char d);  
}
```

```
class IntPuzzle implements IPuzzle {  
    public void scramble() {...}  
    public int tile(int r, int c) {...}  
    public boolean move(char d) {...}  
}
```

- name of interface:  
**IPuzzle**
- a class **implements** this interface by implementing **public instance methods** as specified in the interface
- the class may implement other methods

- An interface is not a class , therefore cannot be instantiated
- A class can implement several interfaces  
    class X implements Ipod, Ipuzzle {...}

# Why interface construct?

- good software engineering
  - specify and enforce boundaries between different parts of a team project
- can use interface as a type
  - allows more generic code
  - reduces code duplication
- Lots of examples in Java

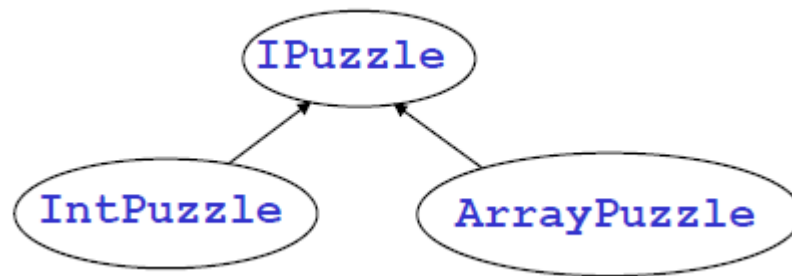
```
Map<String, Command> h = new HashMap<String, Command>();
```

```
List<Object> t = new ArrayList<Object>();
```

```
Set<Integer> s = new HashSet<Integer>();
```

# Example of code duplication

- Suppose we have two implementations of MyList:
  - class **IntList** and **CharList**
- Say the client wants to use both implementations
- client code has a **display method to print out all elements**
- What would the **display method look like?**
  - **write just one display for both types**



- `IPuzzle` → interface, `IntPuzzle` and `ArrayPuzzle` → implementing classes
- interface names can be used in type declarations
  - **`IPuzzle p1, p2;`**
- a class that implements the interface is a subtype of the interface type
  - **`IntPuzzle` and `ArrayPuzzle` are subtypes of `IPuzzle`**
  - **`IPuzzle` is a supertype of `IntPuzzle` and `ArrayPuzzle`**

# What is Inheritance?

- What is Inheritance?
  - a mechanism for Extensibility in OO programming
  - Extensibility: permits behavior of classes to be *changed or extended without having to rewrite the code of the class*  
no need to involve the class implementer
  - promotes code reuse
- Encapsulation: permits code to be used without knowing implementation details
  - classes, objects
  - visibility declarations such as **private**, **protected**
- **OO-programming = Encapsulation + Extensibility**



```
class Puzzle {

    //representation of a puzzle state
    private int state;

    //create a new random instance
    public void scramble() {...}

    //say which tile occupies a given position
    public int tile(int row, int col) {...}

    //move a tile
    public boolean move(char c) {...}
}
```

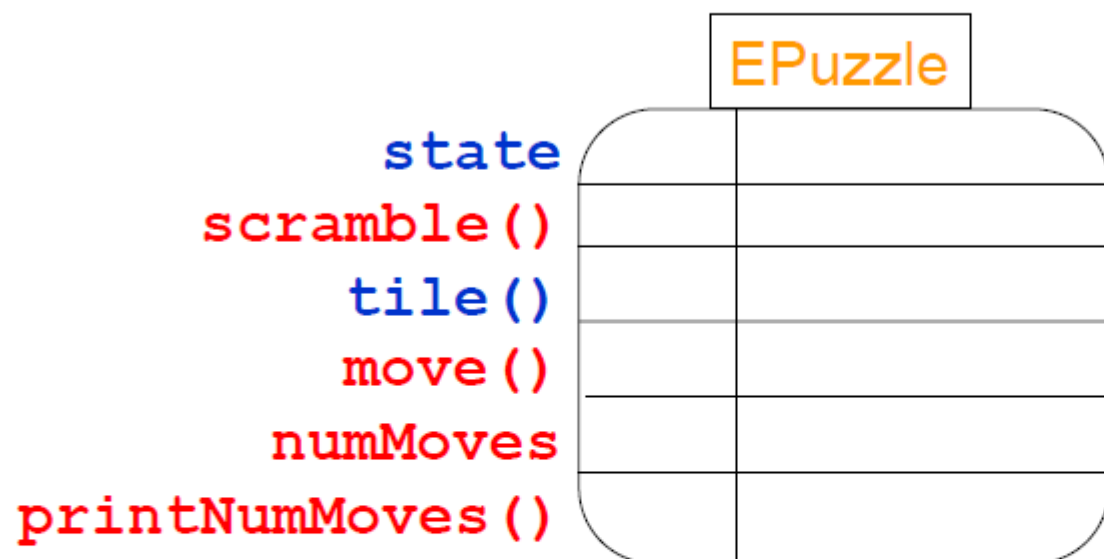
1	2	3
4	2	6
7	5	8

```
class EPuzzle extends Puzzle {
    private int numMoves = 0;
    public void scramble() {...}
    public boolean move(char d) {...}
    public void printNumMoves() {...}
}
```

## Difficulty with Private Variables

- Variable `state` is declared *private*, so it is only accessible to methods in class `Puzzle`
- In an instance of class `EPuzzle`, the `tile` method can access this variable because the `tile` method is *inherited* from the superclass
- Method `scramble` defined in class `Epuzzle` does *not* have access to `state`
- Similarly, any *private* methods in a superclass are not accessible to methods in subclass

# Interesting Point



- `EPuzzle` objects have an instance variable `state` because `EPuzzle` extends `Puzzle`
- However, they cannot access it directly, because it is private!
- `state` is accessible to public methods inherited from `Puzzle` (such as `tile()`) but not to methods written in the `EPuzzle` class (such as `scramble()`)

# Protected Access

- Access specifier: **protected**
- A protected instance field in class **S** can be accessed by instance methods defined in **S** or any subclass of **S**
- A protected method in class **S** can be invoked from an instance method defined in **S** or any subclass of **S**
- Access checks are done by compiler at compile time:
  - For an invocation **x.m()** :
    - Determine the static (compile-time) type of **x**
    - Does the corresponding class/interface have a method named **m** with appropriate arguments?
    - Are the access specifiers of that method appropriate?
- When should variables and methods be declared **protected** instead of **private**?
- Think about extensibility: if subclasses will want access to a member, it should be declared **protected**

# Constructors

- Each class has its own constructor
- No overriding of constructors
- Superclass constructor can be invoked explicitly within subclass constructor using `super()` with parameters as needed
- Can invoke other constructors of the same class using `this()`
- Call to `super()` or `this()` must occur *first* in the constructor

# Exceptions in Java

- **Exceptions** = a mechanism for handling uncommon or exceptional situations
  - unusual, but may be expected from time to time during normal operation
  - example: user enters date in incorrect format
- **Errors** are typically more serious
  - program failure, usually symptom of a bug

# You can create your own exception..

```
class MyException extends Exception {  
    MyException(String message) {  
        super(message) ;  
    }  
}  
  
...  
  
MyException m = new MyException("Bad karma") ;  
System.out.println(m.getMessage()) ;  
  
...
```

# Throwing and Catching

- Certain operations may throw exceptions...

```
int[] a = new int[3];  
a[3] = -1; //throws ArrayIndexOutOfBoundsException  
  
Object o = null;  
String s = o.toString(); //throws NullPointerException  
  
Object e = new Object();  
Integer i = (Integer)e; //throws ClassCastException
```

- ... or you can do it explicitly with **throw**

```
throw new NullPointerException();  
throw new MyException("Bad karma");
```

# Throwing and Catching

```
try {  
    ...code that might throw an exception...  
} catch (NumberFormatException e) {  
    ...exception handler for number format exceptions...  
} catch (ClassCastException e) {  
    ...exception handler for class cast exceptions...  
} catch (Exception e) {  
    ...default handler for all other exceptions...  
}
```



# Throwing and Catching

- What happens when an exception is thrown?
  - Execution of the method is immediately interrupted
  - Control passes to an enabled exception handler if there is one
    - the throw must have occurred in the try clause of a try/catch statement
    - the runtime type of the exception must be a subtype of the type specified in the catch clause
    - the first enabled handler is executed
  - If there is no enabled exception handler, control passes to the calling method and the exception is rethrown at the call point

- Example 1

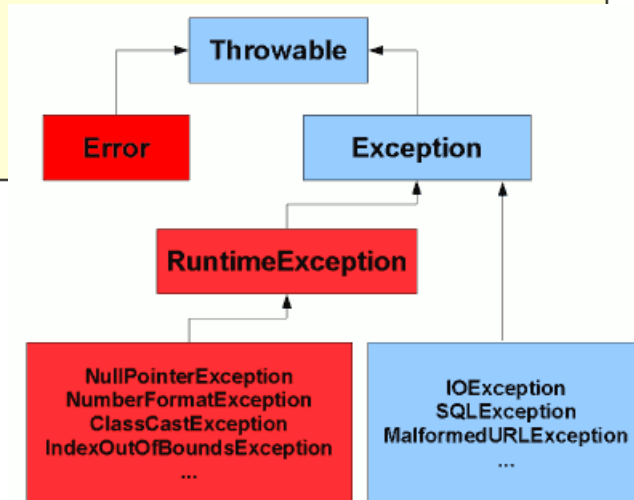
```
try {  
    Object o = new Object();  
    Integer i = (Integer)o;  
} catch (NumberFormatException e) {  
    ...  
} catch (ClassCastException e) {  
    ... //this handler will be executed  
} catch (Exception e) {  
    ...  
}
```

- Example 2

```
try {  
    int[] a = new int[3];  
    a[3] = -1;  
} catch (NumberFormatException e) {  
    ...  
} catch (ClassCastException e) {  
    ...  
} catch (Exception e) {  
    ... //this handler will be executed  
}
```

- Example 3

```
try {  
    //this will not be caught  
    throw new Throwable();  
} catch (NumberFormatException e) {  
    ...  
} catch (ClassCastException e) {  
    ...  
} catch (Exception e) {  
    ...  
}
```



- Example 4

```
try {  
    int i = foo(s);  
} catch (MyException e) {  
    System.out.println(e.getMessage());  
}  
...
```

*All potentially uncaught exceptions (except  
RuntimeExceptions) must be declared*

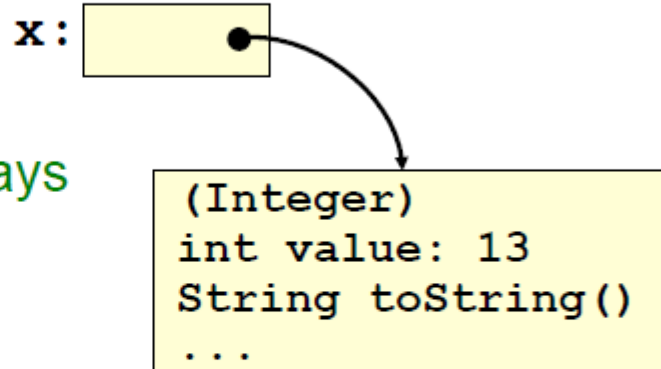
```
int foo(String s) throws MyException {  
    if (s.equals("")) {  
        throw new MyException("Empty argument");  
    }  
}
```

# Extending a Class vs Implementing an Interface

- A class can
  - implement many interfaces, but
  - extend only one class
- To share code between two classes
  - put shared code in a common superclass
  - interfaces cannot contain code

# Reference vs Primitive Types

- Reference types
  - classes, interfaces, arrays
  - E.g.: `Integer`



- Primitive types
  - `int`, `long`, `short`, `byte`, `boolean`, `char`, `float`, `double`

`x:` 13

# Why Both `int` and `Integer`?

- Some data structures work only with reference types
  - (`Hashtable`, `Vector`, `Stack`, ...)
- Primitive types are more efficient
  - `for (int i = 0; i < n; i++) {...}`



# Upcasting and Downcasting

- Applies to reference types only
- Used to assign the value of an expression of one type to a variable of another type
  - upcasting: subtype → supertype
  - downcasting: supertype → subtype
- Example of upcasting: **Object x = new Integer(13);**
  - Integer is a subtype of Object, so this is an upcast
- Example of downcasting: **assume y is of type Object**
  - **Integer x = (Integer)y;**
  - Integer is a subtype of Object, so this is a downcast
  - Compile time check to see left and right side are of same type
  - runtime check, **ClassCastException** if failure
- **Subtyping and upcasting avoids duplicate code!!!!**

## Is the Runtime Check Necessary?

Yes, because dynamic type of object may not be known at compile time

```
void bar() {  
    foo(new Integer(13));  
}  
        String("x")  
  
void foo(Object y) {  
    int z = ((Integer)y).intValue();  
    ...  
}
```

# Method Dispatch

```
public static void display(IPuzzle p) {  
    for (int row = 0; row < 3; row++)  
        for (int col = 0; col < 3; col++)  
            System.out.println(p.tile(row,col));  
}
```

- Which `tile` method is invoked?
  - depends on **dynamic type** of object `p` (`IntPuzzle` or `ArrayPuzzle`)
  - we don't know what it is, but whatever it is, we know it has a `tile` method (since any class that implements `IPuzzle` must have a `tile` method)

# Method Dispatch

```
public static void display(IPuzzle p) {  
    for (int row = 0; row < 3; row++)  
        for (int col = 0; col < 3; col++)  
            System.out.println(p.tile(row,col));  
}
```

- **Compile-time check:** does the **static type** of `p` (namely `IPuzzle`) have a `tile` method with the right type signature? **If not → error**
- **Runtime:** go to **object** that is the value of `p`, find its **dynamic type**, look up its `tile` method
- The compile-time check guarantees that an appropriate `tile` method exists

# Another Use of Upcasting

## Heterogeneous Data Structures

- Example:

```
IPuzzle[] pzls = new IPuzzle[9];  
pzls[0] = new IntPuzzle();  
pzls[1] = new ArrayPuzzle();
```

- expression `pzls[i]` is of type `IPuzzle`
- objects created on right hand sides are of subtypes of `IPuzzle`

# Java instanceof

- Example:  
`if (p instanceof IntPuzzle) {...}`
- true if dynamic type of `p` is a subtype of `IntPuzzle`
- usually used to check if a downcast will succeed

# Subinterfaces

- Suppose you want to extend the interface to include more methods
  - `IPuzzle: scramble, move, tile`
  - `ImprovedPuzzle: scramble, move, tile, samLoyd`
- Two approaches
  - start from scratch and write an interface
  - extend the `IPuzzle` interface

```
interface IPuzzle {  
    void scramble();  
    int tile(int r, int c);  
    boolean move(char d);  
}  
  
interface ImprovedPuzzle extends IPuzzle {  
    void samLoyd();  
}
```

- **IPuzzle** is a superinterface of **ImprovedPuzzle**
- **ImprovedPuzzle** is a subinterface of **IPuzzle**
- **ImprovedPuzzle** is a subtype of **IPuzzle**
- An interface can extend multiple superinterfaces
- A class that implements an interface must implement all methods declared in all superinterfaces



# Array vs ArrayList vs HashMap

- Three extremely useful constructs (see Java API)
- **Array**
  - storage is allocated when array created; cannot change
- **ArrayList** (in `java.util`)
  - an “extensible” array
  - can append or insert elements, access  $i^{\text{th}}$  element, reset to 0 length
  - use with **List** interface
- **HashMap** (in `java.util`)
  - save data indexed by keys
  - can lookup data by its key
  - can iterate over keys or values
  - use with **Map** interface

## HashMap Example

- Create a `HashMap` of numbers, using the names of the numbers as keys:

```
Map<String, Integer> numbers
    = new HashMap<String, Integer>();
numbers.put("one", new Integer(1));
numbers.put("two", new Integer(2));
numbers.put("three", new Integer(3));
```

To retrieve a number:

```
Integer n = numbers.get("two");
```

- returns `null` if the `HashMap` does not contain the key
  - Can use `numbers.containsKey(key)` to check this

# Generics and Autoboxing

- Pre-Java 5

```
Map numbers = new HashMap();  
numbers.put("one", new Integer(1));  
Integer s = (Integer)numbers.get("one");
```

- Java 5 (with generics)

```
Map<String, Integer> numbers =  
    new HashMap<String, Integer>();  
numbers.put("one", new Integer(1));  
Integer s = numbers.get("one");
```

- Java 5 (with generics + autoboxing)

```
Map<String, Integer> numbers =  
    new HashMap<String, Integer>();  
numbers.put("one", 1);  
int s = numbers.get("one");
```

# Coding and Debugging Advice

- Don't be afraid to experiment if you are not sure how things work
  - Documentation isn't always clear
  - *Interactive Development Environments (IDEs)*, e.g. Eclipse, make this easier
- Debugging
  - Do not just make random changes, hoping something will work
  - Think about what could cause the observed behavior
  - Isolate the bug – use print statements
  - An IDE makes this easier by providing a *Debugging Mode*
    - Can set breakpoints, step through the program while watching chosen variables