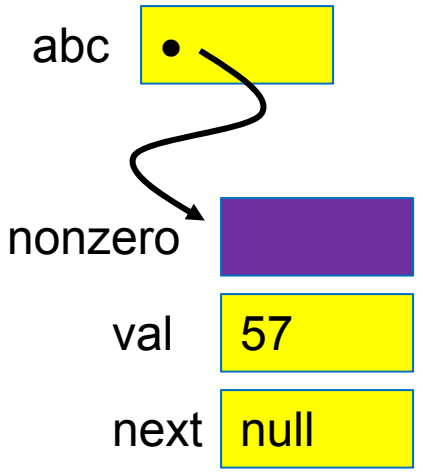
MORE ON SUBCLASSES, INHERITANCE, INTERFACES, ETC

Lecture 4
CS2110 – Fall ‘10

Primitive vs Reference Types

- Primitive types
  - int, short, long, float, byte,
  - char, boolean, double
- Efficient
  - 1 or 2 words
  - Not an Object—unboxed

- Reference types
  - Objects and arrays
  - String, int[], HashSet
  - Usually require more memory
  - Can have special value null
  - Can compare null with ==, !=
  - Generates NullPointerException if you try to dereference null



Comparing/copying primitive types

- Works just as you would expect

```
int a, b;

if(a < b) { … }

a = b+3;
```

Comparing/Copying Reference Types

- Comparing objects (or copying them) isn't easy!
  - You need to copy them element by element
  - Compare objects using the "equals" method, which implements "deep equality"

What you wrote	How to write it correctly
"xy" == "xy"	"xy".equals("xy")
"xy" == "x" + "y"	"xy".equals("x" + "y")
"xy" == new String("xy")	"xy".equals(new String("xy"))

Inheritance

- A subclass *inherits* the methods of its superclass
- Example: methods of the Object superclass:
  - equals(), as in A.equals(B)
  - toString(), as in A.toString()
  - … others we'll learn about later in the course
- … every object thus supports toString()!

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 actual, underlying object

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 (declared) type of the variable r, not the runtime type of the object referred to by r
- Shadowing variables is bad medicine and should be avoided

… a nasty example

```
class A {
    int i = 1;
    int f() { return i; }
}
class B extends A {
    int i = 2;                          // Shadows variable i in class A.
    int f() { return -i; }              // Overrides method f in class A.
}
public class override_test {
    public static void main(String args[]) {
        B b = new B();
        System.out.println(b.i);        // Refers to B.i; prints 2.
        System.out.println(b.f());      // Refers to B.f(); prints -2.
        A a = (A) b;                    // Cast b to an instance of class A.
        System.out.println(a.i);        // Now refers to A.i; prints 1;
        System.out.println(a.f());      // Still refers to B.f(); prints -2;
    }
}
```

The “runtime” type of “a” is “B”!

… a nasty example

```
class A {
    int i = 1;
    int f() { return i; }
}
class B extends A {
    int i = 2;                          // Shadows variable i in class A.
    int f() { return -i; }              // Overrides method f in class A.
}
public class override_test {
    public static void main(String args[]) {
        B b = new B();
        System.out.println(b.i);        // Refers to B.i; prints 2.
        System.out.println(b.f());      // Refers to B.f(); prints -2.
        A a = (A) b;                    // Cast b to an instance of class A.
        System.out.println(a.i);        // Now refers to A.i; prints 1;
        System.out.println(a.f());      // Still refers to B.f(); prints -2;
    }
}
```

The “declared” or “static” type of “a” is “A”!

Interfaces

- What is an interface? Informally, it is a specification of how an object interacts with the outside world

- Java has a construct called **`interface`** which is used formally for this purpose
  - an interface describes how a class interacts with its clients
  - method names, argument/return types, fields

Inheritance and Overriding let us create families of related classes

- For example:
  - Sets
  - Array is a primitive reference type
  - ArrayList is a subclass of Set and implements the Array interface
  - HashMap is a subclass of Map and implements the Array interface

- All of these classes support similar functionality because they offer the same “interface” and interpret the operations in the same way
- But they are implemented differently in support of different styles of use

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

Notes

- An interface is not a class!
  - cannot be instantiated
  - incomplete specification

- class header must assert **`implements I`** for Java to recognize that the class implements interface **`I`**

- A class may implement several interfaces:

```
class X implements IPuzzle, IPod {...}
```

Why an `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

Why an interface construct?

- 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 puzzles:
  - class **`IntPuzzle`** uses an **`int`** to hold state
  - class **`ArrayPuzzle`** uses an array to hold state
- Say the client wants to use both implementations
  - perhaps for benchmarking both implementations to pick the best one
  - client code has a **`display`** method to print out puzzles
- What would the **`display`** method look like?

```
class Client{
  IntPuzzle p1 = new IntPuzzle();
  ArrayPuzzle p2 = new ArrayPuzzle();
    ...display(p1)...display(p2)...

  public static void display(IntPuzzle p){
    for (int r = 0; r < 3; r++)
      for (int c = 0; c < 3; c++)
        System.out.println(p.tile(r,c));
  }

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

Code duplicated because **`IntPuzzle`** and **`ArrayPuzzle`** are different

Observation

- Two display methods are needed because **`IntPuzzle`** and **`ArrayPuzzle`** are different types, and parameter **`p`** must be one or the other

- but the code inside the two methods is identical!
  - code relies only on the assumption that the object **`p`** has an instance method **`tile(int,int)`**

- Is there a way to avoid this code duplication?

One Solution — Abstract Classes

Puzzle code

```
abstract class Puzzle {
  abstract int tile(int r, int c);
  ...
}
class IntPuzzle extends Puzzle {
  public int tile(int r, int c) {...}
  ...
}
class ArrayPuzzle extends Puzzle {
  public int tile(int r, int c) {...}
  ...
}
```

Client code

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

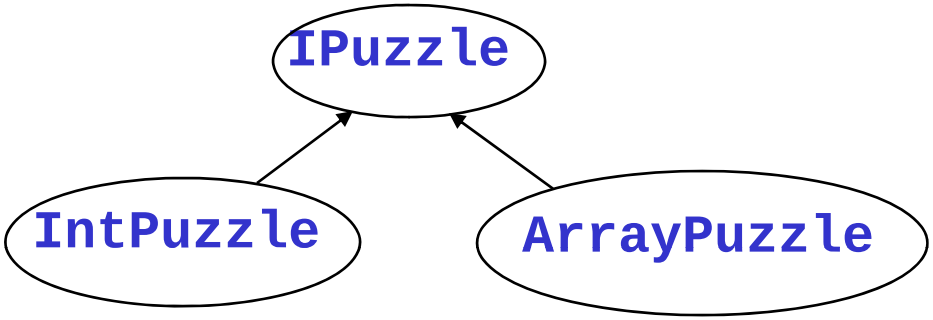
Another Solution — Interfaces

Puzzle code

```
interface IPuzzle {
  int tile(int r, int c);
  ...
}
class IntPuzzle implements IPuzzle {
  public int tile(int r, int c) {...}
  ...
}
class ArrayPuzzle implements IPuzzle {
  public int tile(int r, int c) {...}
  ...
}
```

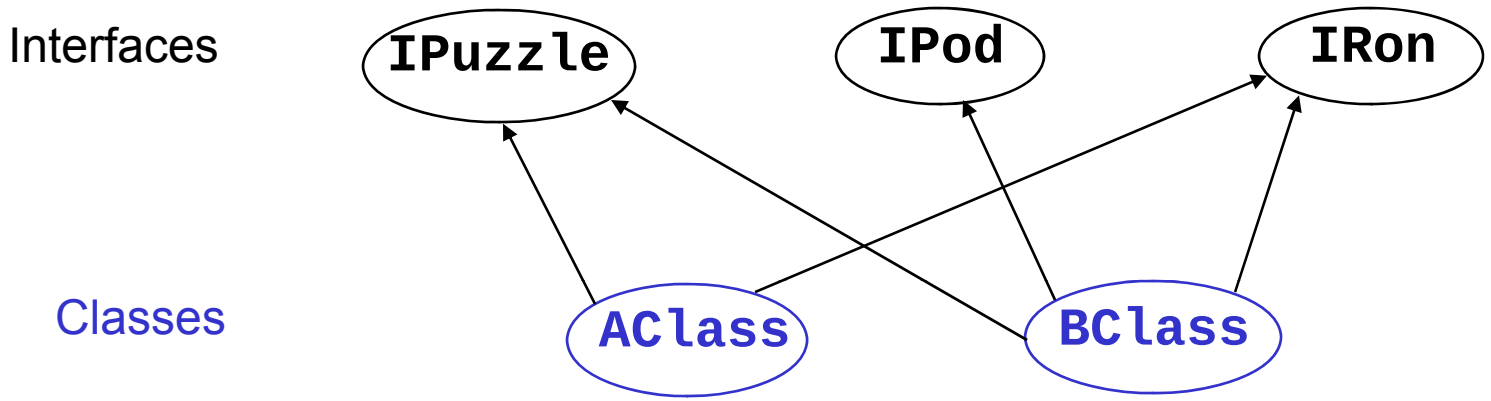
Client code

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



- 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`**



- Unlike classes, types do not form a tree!
  - a class may implement several interfaces
  - an interface may be implemented by several classes

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

Static vs Dynamic Types

- Every variable (more generally, every expression that denotes some kind of data) has a static* or compile-time type
  - derived from declarations – you can see it
  - known at compile time, without running the program
  - does not change

- Every object has a dynamic or runtime type
  - obtained when the object is created using **new**
  - not known at compile time – you can't see it

---

* Warning! No relation to Java keyword **static**

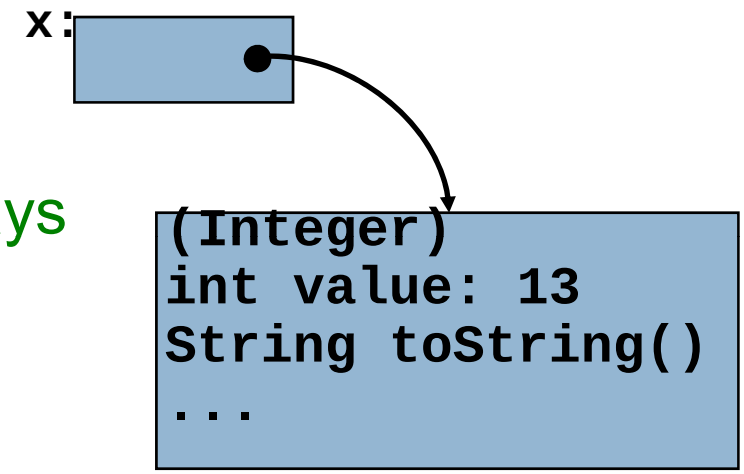
Example


```
int i = 3, j = 4;
Integer x = new Integer(i+3*j-1);
System.out.println(x.toString());
```

- static type of the variables `i,j` and the expression `i+3*j-1` is `int`
- static type of the variable `x` and the expression `new Integer(i+3*j-1)` is `Integer`
- static type of the expression `x.toString()` is `String` (because `toString()` is declared in the class `Integer` to have return type `String`)
- dynamic type of the object created by the execution of `new Integer(i+3*j-1)` is `Integer`

Reference vs Primitive Types

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



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

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 (static) type to a variable of another (static) type
  - upcasting: subtype □ supertype
  - downcasting: supertype □ subtype
- A crucial invariant:

If during execution, an expression *E* is ever evaluated and its value is an object *O*, then the dynamic type of *O* is a subtype of the static type of *E*.

Upcasting

- Example of upcasting:

```
Object x = new Integer(13);
```

  - static type of expression on rhs is **`Integer`**
  - static type of variable **`x`** on lhs is **`Object`**
  - **`Integer`** is a subtype of **`Object`**, so this is an upcast

- static type of expression on rhs must be a subtype of static type of variable on lhs – compiler checks this

- upcasting is always type correct – preserves the invariant automatically

Downcasting

- Example of downcasting:

```
Integer x = (Integer)y;
```

  - static type of **`y`** is **`Object`** (say)
  - static type of **`x`** is **`Integer`**
  - static type of expression **`(Integer)y`** is **`Integer`**
    - **`Integer`** is a subtype of **`Object`**, so this is a downcast
- In any downcast, dynamic type of object must be a subtype of static type of cast expression
- runtime check, **`ClassCastException`** if failure
- needed to maintain invariant (and only time it is needed)

Some type checking can only be done at runtime

…. 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();
  ...
}
```

Upcasting with Interfaces

- Java allows up-casting:

```
IPuzzle p1 = new ArrayPuzzle();
IPuzzle p2 = new IntPuzzle();
```

- Static types of right-hand side expressions are **`ArrayPuzzle`** and **`IntPuzzle`**, resp.

- Static type of left-hand side variables is **`IPuzzle`**

- Rhs static types are subtypes of lhs static type, so this is ok

Why Upcasting?

- Subtyping and upcasting can be used to avoid code duplication
- Puzzle example: you and client agree on interface **`IPuzzle`**

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

Solution

Puzzle code

```
interface IPuzzle {
  int tile(int r, int c);
  ...
}
class IntPuzzle implements IPuzzle {
  public int tile(int r, int c) {...}
  ...
}
class ArrayPuzzle implements IPuzzle {
  public int tile(int r, int c) {...}
  ...
}
```

Client code

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

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

Note on Casting

- Up- and downcasting merely allow the object to be viewed at compile time as a different static type
- Important: when you do a cast, either up or down, nothing changes
  - not the dynamic type of the object
  - not the static type of the expression

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
- When is this useful?
  - Enables us to write “reflexive” code: software that operates in very general ways and customizes its behavior based on the types of objects it “observes”

Example

- suppose **`twist`** is a method implemented only in **`IntPuzzle`**

```
void twist(IPuzzle[] pzls) {
  for (int i = 0; i < pzls.length; i++) {
    if (pzls[i] instanceof IntPuzzle) {
      IntPuzzle p = (IntPuzzle)pzls[i];
      p.twist();
    }
  }
}
```

Avoid Useless Downcasting

bad

```
void moveAll(IPuzzle[] pzls) {
  for (int i = 0; i < pzls.length; i++) {
    if (pzls[i] instanceof IntPuzzle)
      ((IntPuzzle)pzls[i]).move("N");
    else ((ArrayPuzzle)pzls[i]).move("N");
}}
```

good

```
void moveAll(IPuzzle[] pzls) {
  for (int i = 0; i < pzls.length; i++)
    pzls[i].move("N");
}
```

Subinterfaces

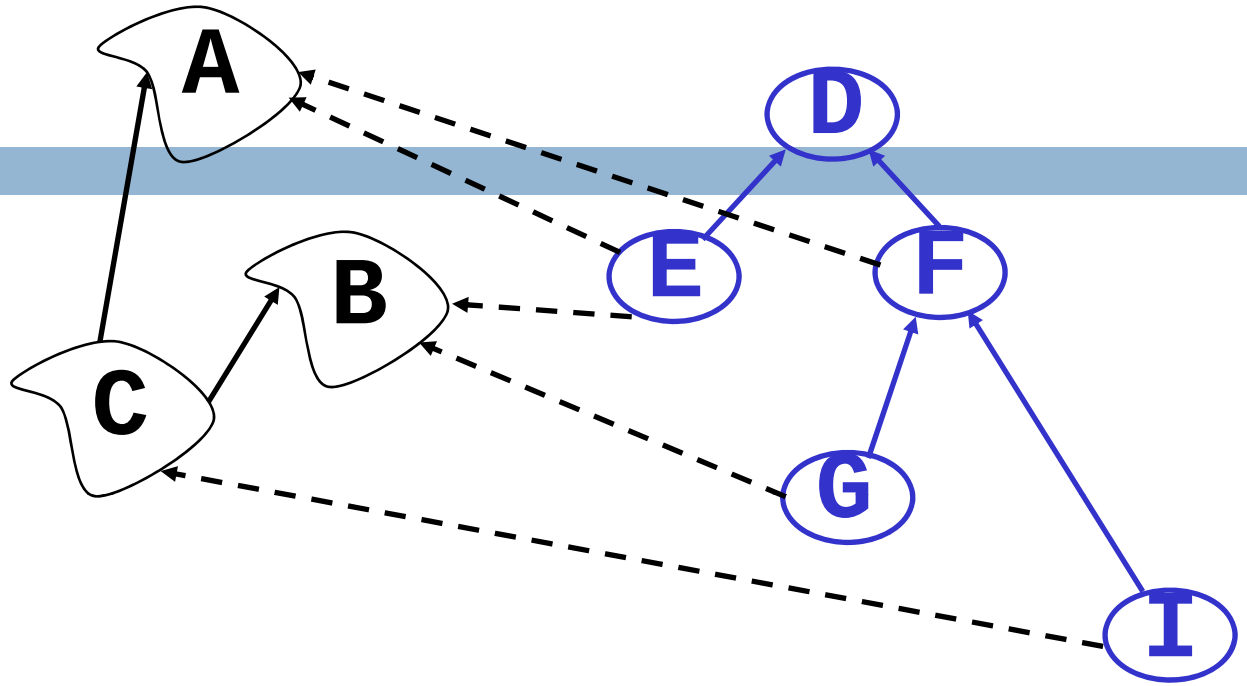
- Suppose you want to extend the interface to include more methods
  - **`IPuzzle`**: **`scramble, move, tile`**
  - **`ImprovedPuzzle`**: **`scramble, move, tile, samLoyd`**

- 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




```
interface C extends A,B {...}
class F extends D implements A {...}
class E extends D implements A,B {...}
```

Conclusion

- Relationships between classes are a “tool” in Java
  - This tool lets us, for example, talk about “Living creatures”, “all animals” “animals in the Bronx zoo”, “Lenny the Lion”, etc.
  - Java is sophisticated about these relationships: subclasses, inheritance, interfaces, overriding, shadowing… We need to understand these mechanisms to use Java well.
- But we also need to use them carefully!
  - Very easy to create confusing situations!