**Inheritance, or extends**

- Inheritance and Java interfaces (more on this later) are the key ideas for achieving reuse of software components in Java.

- Basic idea: Given a class A, define a new class B based on A but with some differences too.
  - We call A the parent or base class.
  - We call B the child or derived class.

- By default, B inherits all of the variables and methods of A.

- The class definition of B just describes the differences from A:
  - B can add new variables and methods.
  - B can override (i.e., redefine) the parent’s methods.

```java
class ExtStack extends stack {
    // Create an ExtStack of using the default size.
    public ExtStack() {
        // Call to super() is implicit.
        System.out.println("Creating an ExtStack!");
    }

    // Create an ExtStack of a specified size.
    public ExtStack(int size) {
        // super() will be called if we leave out the next
        // line; we want the analogous constructor instead!
        super(size);
        System.out.println("Creating an ExtStack of size " + size + ");
    }

    // Add a print statement and call parent routine.
    public void push (Object elt) {
        System.out.println("Pushing onto an ExtStack.");
        super.push(elt);
    }

    // Interchange top two elts on the stack.
    public void swap () {
        final Object e1 = this.pop();
        final Object e2 = this.pop();
        this.push(e1);
        this.push(e2);
    }

    // Return the top element without removing
    // it permanently.
    public Object peek () {
        final Object e1 = this.pop();
        push (e1);
        return e1;
    }

    // Simple test driver for this class.
    public static void main (String [] args) {
        ExtStack s1 = new ExtStack();
        stack s2, s3;
        s2 = new ExtStack(100);
        s3 = new stack();
        int i;
        for (i=0; i<10; i++) {
            s1.push(new Integer(i));
        }
        s1.print();
        System.out.println("Last element is " + s1.peek());
        s1.swap();
        System.out.println("After swap, last elt is " + s1.peek());
        s1.swap();
        System.out.println("After another swap, last elt is " + s1.peek());
        s1.print();
        System.out.println("Done!");
    }
}
```

---

**Single Inheritance**

- defines a tree-like hierarchy on classes (actually a partial order)

```
(Single) Inheritance
```

- Can instantiate or set p to point to an object of class A or any descendant of A.
- Can instantiate or set q to point to an object of class C or any descendant of C, but q may not be instantiated to objects of classes A, B or D.
Static vs. Dynamic Types

Static type:

```
A p;
B q;
```

The declaration sets the static type of a pointer/reference.a

→ The static type remains unchanged throughout its life.

→ Here, \( p \) has static type "reference to \( A \)",
and \( q \) has static type "reference to \( B \)".

→ The static type can be determined by just looking at the source code!

We use the terms "pointer" and "reference" interchangeably. Some people object to this usage.

Dynamic type:

```
p = new D();
q = p;
```

This run-time bind sets the dynamic type of \( p \) and \( q \).

→ The dynamic type is reset every time a new or an assignment is done.

→ A pointer may have several different dynamic types throughout its lifespan (one at a time!).

→ The dynamic type can only be determined by asking the system while the program is executing, "What kind of object are you currently pointing to?" [Java "reflection" is the more general idea.]

→ An object reference that is a formal parameter may have a different dynamic type with each invocation of the function. [More on this later.]

→ The dynamic type must be an inheritance descendant of the static type. In the above example, \( p \) is given the dynamic type "reference to \( D \)".

Static vs. Dynamic Types (cont’d)

In a nutshell:

**static type** — what the reference was declared to be a pointer to.

**dynamic type** — the class of the object that the reference is currently pointing to.

The dynamic type must be an inheritance descendent of (or the same as) the static type.

Consider:

```
stack s4;
s4 = new ExtStack ();
...
s4.swap();
```

The last line is illegal and won’t compile. Why?

In Java (and C++), you may only refer to methods/variables of the static type of a reference

\[ \Rightarrow \text{cannot call } s4.swap() \]

If you want to be able to access the properties of the dynamic type (the class of the actual object), you have to perform a co-ercion.

There are two common ways to do this:

1. [Simple] Perform a "cast"a:

   ```
   stack s4;
s4 = new ExtStack ();
   ...
   ExtStack s5;
s5 = (ExtStack) s4;
s5.swap();
   ```

   An illegal cast will cause a run-time exceptionb to be raised.

2. [More powerful and more complicated.]
   Use Java "reflection".

   aThis is safe in Java; it’s unsafe in C/C++!
   bjava.lang.ClassCastException
Java “Reflection”

The reflection features allow you to ask Java objects at run-time questions like:

- “Say, just what kind of thing are you referencing right now?”
- “Can I have another one just like you?”
- “Please create an instance of the class whose name is in this string variable.”
- “Who does your defining class inherit from?”
- “Which interfaces do you implement?”

Simple use:

```java
if (e2 instanceof ExtStack) {
    :
}
```

Java version 1.1 adds significantly more powerful reflection features in `java.lang.reflect`.

- This is darned useful stuff, especially when you dynamically load libraries and then need to enquire about the kinds of entities you’re dealing with!

- The Java class `Object` (which all classes inherit from) defines a method called `getClass` which returns an object of type `java.lang.Class` which defines methods such as `newInstance`, `getName`, `getSuperClass` etc.

Here’s an example of how to instantiate when the name of the class is stored in a String (might be read in from a text file):

```java
try {
    Class extStackClass = Class.forName("ExtStack");
    ExtStack s6 = (ExtStack) extStackClass.newInstance();
    s6.push("Hello");
    System.out.println(s6.peek());
} catch (Exception excpt) {
    System.out.println(excpt);
}
```

Java Constructor “Chaining”

- Java guarantees that if you create an instance of class `foo`, then a constructor of its superclass is also called as the first action (and so on transitively up the inheritance hierarchy).
- You may decide which superclass constructor you wish to call yourself and place it as the first statement in each constructor.
- ... or if the first statement in a constructor is not of the form `super(...)`, then by default Java will call the parent constructor of no arguments `super()`.
- Moral: Remember when you define a constructor in a derived class to put in a call to the appropriate super constructor (or it will be done for you).

Inheritance and Polymorphism

Problem:

- have several related but distinct kinds of objects.
- want a simple, uniform way of modelling this situation.

OO Solution:

```
use inheritance
```

![Diagram of inheritance hierarchy]

- Circle
- Rectangle
- Square
abstract class Figure {
    // Location of Figure on screen.
    protected int x=0, y=0;

    // Accessor function to set the location.
    public void setLoc (int newX, int newY) {
        x = newX;
        y = newY;
    }

    // Details of these will depend on children.
    public abstract void draw ();
    public abstract double area ();
    public abstract void setSize (int newSize);
    public abstract void setSize (int newW, int newH);

    // Useful aux. function used by descendants.
    protected String myClassName () {
        return this.getClass().getName();
    }
}

class Circle extends Figure {
    // The radius of the circle.
    protected int radius=0;

    // Should do some graphics stuff here,
    // but let's keep the examples simple for now.
    public void draw () {
        System.out.println (myClassName() +
                        " at x = " + x + " y = " + y +
                        " with radius = " + radius);
    }

    // Calculate the area of the circle.
    public double area () {
        return Math.PI * (double) (radius * radius);
    }

    // setSize of one arg sets the radius.
    public void setSize (int newSize) {
        radius = newSize;
    }

    // setSize of two args makes no sense here.
    public void setSize (int newW, int newH) {
        System.err.println("Can't setSize of two "+
                          "args with Circles");
    }
}

class Rectangle extends Figure {
    // The width and height of the Rectangle
    protected int w=0, h=0;

    // Should do some graphics stuff here,
    // but let's keep the examples simple for now.
    public void draw () {
        System.out.println (myClassName() + " a tx=" + x + " y=" + y +
                        " with width = " + w + " and height = " + h);
    }

    // setSize of two args sets width and height resp.
    public void setSize (int newW, int newH) {
        w = newW;
        h = newH;
    }

    // setSize of one arg makes no sense for
    // normal Rectangles
    public void setSize (int newSize) {
        System.err.println("Can't setSize of "+
                          "one arg with Rectangles");
    }

    // return the area of a Rectangle/
    public double area () {
        return (double) (w * h);
    }
}

class Square extends Rectangle {
    // setSize of one arg sets the width
    // and height to this value.
    public void setSize (int newSize) {
        w = newSize;
        h = newSize;
    }

    // setSize of two args on Squares sets the
    // width and height to the greater of the two
    // provided arg values.
    public void setSize (int newW, int newH) {
        final int max = Math.max (newW, newH);
        w = max;
        h = max;
    }
}
At the top level of the hierarchy (i.e., Figure), some methods are declared as abstract.

```
public abstract void draw();
```

→ No provided body; just a semi-colon to end the line.
→ The intent is that subsequent derived classes will define this function appropriately, but there is no obvious “default” behaviour for draw().

A Java class that has any deferred methods is called an abstract class and must be declared as such.

→ Not all methods need to be abstract in an abstract class! Sometimes, you can define generic methods that will work for descendants e.g., setLoc
→ In Java, can’t instantiate from an abstract class.

Sidebar Why is the Cast Needed?

```
while (!figureStack.isEmpty()){
    Figure f;
    f = (Figure) figureStack.pop();
    f.draw();
}
```

Look at signature for pop() in our class stack:

```
public Object pop () ...
```

We have defined our stack to be as general as possible; can push and pop any instance of Object or any inheritance descendant [i.e., any instance of any class]

pop promises only to return an Object; doesn’t tell us a lot about what kind of entity we are getting back.

If we want to make sure it’s a Figure (or descendant), then declare “receiving” reference to be a Figure; need a cast to make sure all is OK.

Java will throw an exception if popped element is of wrong type.
### Extending abstract Classes

- If you extend an abstract class then either:
  1. you must provide implementations for all of the abstract methods you inherited, or
  2. you must declared the class to be abstract too.

- It is legal in Java for the child to make a method/variable less protected than in the parent, but you may not make it more protected.
  
  → Why? The answer is complicated ...
  
  → It is best simply never to change the protection level (public, protected, etc.) of an inherited method or variable.

- Polymorphism is a very powerful concept:
  
  → The base class defines the shape of all objects in the hierarchy, and the derived classes provide appropriate implementations.

### Java interfaces

- A Java interface is basically a class with no variables and only abstract methods.

- An interface $A$ may be implemented by a class $B$.

  ```java
class B extends C implements A
```

  → $B$ must implement all of $A$’s methods or else be declared as abstract.

  → $B$ can also extend a class, say $C$.

- A class can implement multiple interfaces, but can extend only one class.

  `[java.lang.Object by default, if no other class specified.]`

- An interface can extend other interfaces (any number).

- An interface cannot extend or implement a class.

- Java interfaces are subtle and powerful constructs. More later.

\(^{a}\)Actually, `final` variables (i.e., constants) are allowed.

### Java Wrapper Classes

- In Java, if a class doesn’t explicitly extend another class, then it is assumed to extend the class `java.lang.Object`

- Therefore, to create a truly generic stack class, use `Object` as the element type.

- **Problem**: The following basic types are not classes per se:
  ```
  int, long, short, byte, boolean, double, float, char
  [Sometimes, we call these scalar types.]
  ```

- How do we put ints on a stack?

  **Answer**: **Wrapper classes**. For those times when you want to treat ints, etc. as objects, use the appropriate wrapper class:

  ```
  Integer, Long, Short, Byte, Boolean, Double, Float, Character
  ```

  These are called **wrapper classes** as they basically just wrap a simple value into a class so it can be treated as an object.

### An Example

```java
stack s1 = new stack();
s1.push (new Integer(37));
int i = -13;
s1.push (new Integer(i));
s1.push (new Double (Math.PI));
Double d = (Double) s1.pop();
double halfPI = d.doubleValue() / 2;
System.out.println("P i=",d);
System.out.println("half pi = " + halfPI);
Integer i1 = (Integer) s1.pop();
System.out.println ("Next value is " + i1);
i1 = (Integer) s1.pop();
int j = i1.intValue();
System.out.println ("Next value is " + j);
```
Genericity

What if we want stacks of different kinds of objects, e.g., Strings, integers, Figures, etc?

Approach #1:

- Define a separate class for each kind of element:

```java
class StringStack {
    public void push (String s) {...}
    public String pop () {...}
}
class FigureStack {
    public void push (Figure f) {...}
    public Figure pop () {...}
}
class IntegerStack {
    public void push (Integer i) {...}
    public Integer pop () {...}
}
```

Advantages:

- Can guarantee that a FigureStack will only ever hold instances of Figures (and descendants).
- Can’t accidentally add a String to a FigureStack! This is a good thing!

Disadvantages:

- Cumbersome and tedious. Far too much work for such a simple idea.

Approach #2

- Build general purpose stack that can handle any type of object (i.e., what we did already)
- Use Object as the element type, since any Java object can be treated as an instance of Object and any scalar can be converted into one using a wrapper class.

```java
class stack {
    public void push (Object e) {...}
    public Object pop () {...}
}
```

Advantages:

- We get a truly generic stack!
- Can deal with any kind of object.
- Can convert scalars (e.g., int, boolean, etc.) into objects using wrapper classes.

Disadvantages:

1. Less safety: cannot specify that you want stack to contain only Figures. Client can always put any kind of object on it (accidentally or not).
   → Can fix this with a bit of extra effort using reflection.
2. Client needs to cast back to correct type when you retrieve an element.

```java
Figure f = (Figure) s1.pop();
```
What Does an Object Look Like?

(from the Java API documentation)

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

clone() Creates a new object of the same class as this object.
equals(Object) Compares two Objects for equality.
finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
getClass() Returns the runtime class of an object.
hashCode() Returns a hash code value for the object.
notify() Wakes up a single thread that is waiting on this object's monitor.
notifyAll() Wakes up all threads that are waiting on this object's monitor.
toString() Returns a string representation of the object.
wait() Waits to be notified by another thread of a change in this object. [3 versions of this.]

Approach #3:

- Would like to define a single generic stack class, pass in the desired element type at creation. Code would guarantee that only objects of that element type (and its descendants) can be added to the stack.

- In C++, this is done via a technique called templates. There is a big library of generic data structures called the Standard Template Library (STL).

- In Java, can implement this using reflection.

```java
class genericStack {
    protected Class elementType;
    ...
    public genericStack (Class desiredEltType) {
        maxSize = DefaultMaxSize;
        store = new Object[maxSize];
        elementType = desiredEltType;
    }
    public void push (Object newVal) throws Exception {
        if (!elementType.isInstance(newVal)) {
            throw new Exception("Expected element "+
                        "of type " + elementType
                        "but got element of type "
                        + newVal.getClass());
        } else if (top < maxSize -1) {
            top++;
            store[top]=newVal;
            numValidPushes++;
        } else {
            System.err.println("Sorry, stack is full.");
        }
    }
    ...
    // rest of class unchanged from "stack"
}
```

```java
public static void main (String [] args) {
    try {
        genericStack stk1 = new genericStack
                (Class.forName("java.lang.String"));
        stk1.push("hello");
        stk1.push("there");
        stk1.push("world");
        String s = (String) stk1.pop();
        System.out.println("popped "+s+"\n");
        s = (String) stk1.pop();
        System.out.println("popped "+s+"\n");
        s = (String) stk1.pop();
        System.out.println("popped "+s+"\n");
        // Should be empty.
        s = (String) stk1.pop();
        // Should be caught as an error.
        stk1.push (new Integer(37));
    } catch (Exception e) {
        System.out.println (e);
    }
}
```
Advantages:

- Truly generic stack, plus safety of knowing all pushes will check element type.

Disadvantages:

- Still have to cast back to Figure/String/etc. on pop.

```java
Figure f = (Figure) s1.pop();
```

- Pretty clunky, lots of extra non-obvious code to wrap around.

- Our example requires way too much exception handling for simple use.
  [Can redesign ‘tho.]

**Moral:** Suggest you use Approach #2 until further notice.