CS3101-3 Programming Language – Java

Fall 2004

Sept. 22

Road map today

- Brief review
- Details of class
  - Constructor
  - ***this*** reference
  - Inheritance
  - Overloading
  - Dynamic binding
- Interface
- Exceptions

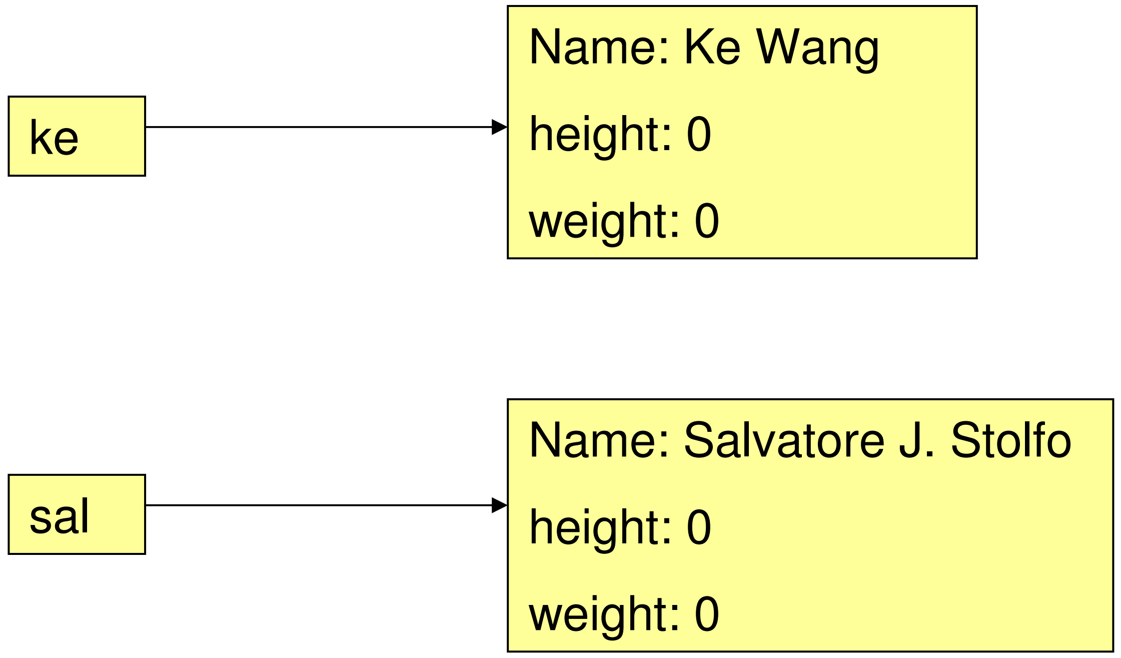
What is Java

- A programming language
- A virtual machine – JVM
- A runtime environment – JRE
    - Predefined libraries
- Portable, but slow
    - Interpreter
    - JIT helps

Object and class

- A class is a blueprint
- An object is an instance created from that blueprint
- All objects of the same class have the same set of attributes
  - Every Person object have name, weight, height
- But different value for those attributes
  - ke.name=Ke Wang, sal.name=Sal Stolfo

Class Person: illustration



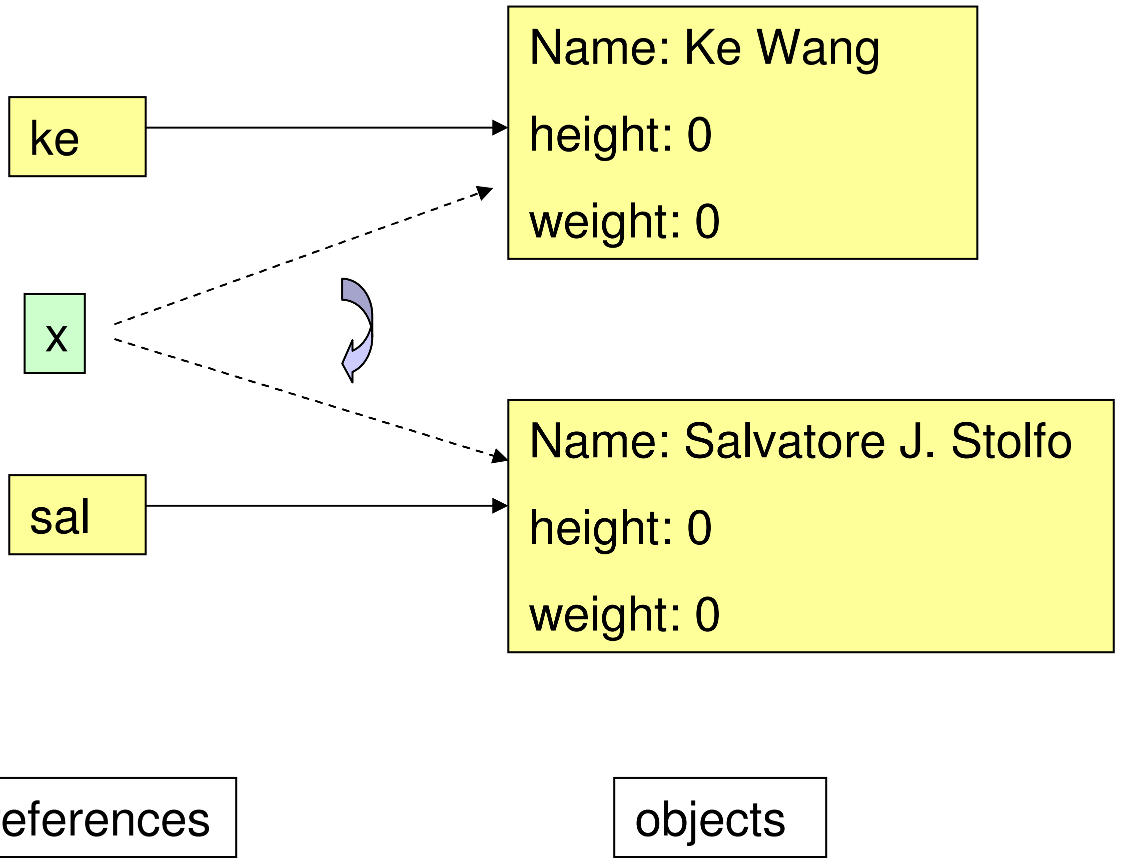
Reference

```
Person ke;              //only created the reference, not an object.
                        It points to nothing now (null).

ke = new Person();      //create the object (allocate storage
                        in memory), and ke is initialized.

ke.name="Ke Wang";      //access the object through
                        the reference
```

Can have multiple reference to one object

No reference means the object is inaccessible forever – goes to garbage collector

Class Person: variables



Visibility of fields and methods

- Generally make fields **private** and provide **public** getField() and setField() accessor functions
- O-O term: encapsulation
- Private fields and methods cannot be accessed from outside of the class.

Static vs. non-static

- Static: class variable/method
- Non-static: instance variable/method
- Static ones are associated with class, not object. Can be called using class name directly
- main() is static
  - Even though it's in a class definition, no instance of the class exist when main starts executing

Static vs. non-static (cont.)

- Instance fields define an object; the values of those fields make one object distinct from another
- Instance method operates on an instance of a class (object) instead of operating on the class itself.
- Class methods can only use class fields; while instance methods can use both instance fields and class fields

How instance method works?

```
Person a=new Person(), b=new Persion();
a.setWeight(100);    b.setWeight(120);
```

- How can the method know whether it’s been called for object a or b?
  - Internal: Person.setWeight(a, 100);
- Invisible additional parameter to all instance methods: ***this***
- It holds a reference to the object through which the method is invoked
  - a.setWeight(100) ➔ this=a

Keyword this

- Can be used only inside method
- When call a method within the same class, don’t need to use ***this***, compiler do it for you.
- When to use it?
  - method parameter or local variable in a method has the same name as one of the fields of the class
  - Used in the return statement when want to return the reference to the current object.
- Example …

Keyword *this* example I

```
class A{
        int w;
        public void setValue (int w) {
                this.w = w;  //same name!
        }
}
```

When a method parameter or local variable in a method has the same name as one of the fields of the class, you must use this to refer to the field.

Keyword *this* example II

```
class Exp{
        public int i=0;
        public Exp increment () {
                i++;
                return this; // return current object
        }

        public static void main (String[] args){
                Exp e = new Exp();
                int v = e.increment().increment().i; // v=2!!
        }
}
```

Object life cycle

- Life cycles of dynamically created objects
- C
  - alloc() – use – free()
- C++
  - new() – constructor() – use – destructor()
- Java
  - new() – constructor() – use – [ignore / garbage collection]

Constructor

- A special method automatically called when an object is created by new()
- Java provide a default one that takes no arguments and perform no special initialization
  - Initialization is guaranteed
  - All fields set to default values: primitive types to 0 and false, reference to null

Constructor (cont.)

- Must have the same name as the class name
  - So the compiler know which method to call
- Perform any necessary initialization
- Format: *public ClassName(para){…}*
- No return type, even no void!
  - It actually return current object
- Notice: if you define any constructor, with parameters or not, Java will not create the default one for you.

Constructor example

```
class Circle{
        double r;
        public static void main(String[] args){
                Circle c2 = new Circle(); // OK, default constructor
                Circle c = new Circle(2.0); //error!!
        }
}
```

Constructor example

```
class Circle{
        double r;
        public Circle (double r) {
                this.r = r;  //same name!
        }
        public static void main(String[] args){
                Circle c = new Circle(2.0); //OK
                Circle c2 = new Circle(); //error!!, no more default
        }
}
```

```
Circle.java:8: cannot resolve symbol
symbol  : constructor Circle ()
location: class Circle
                Circle c2 = new Circle(); //error!!
                            ^
1 error
```

Constructor example

```
class Circle{
            double r;
            public Circle(){
                        r = 1.0; //default radius value;
            }
            public Circle (double r) {
                        this.r = r;  //same name!
            }
            public static void main(String[] args){
                        Circle c = new Circle(2.0); //OK
                        Circle c2 = new Circle();    // OK now!
            }
}
```

Multiple constructor now!!

Method overload

- It’s legal for a class to define more than one method with the **same name**, as long as they have **different list of parameters**
  - Different number of parameter, or different type of parameter, or different order
  - Must be the same return type
  - The method can be static or not, or both: some are static, some are not.
- The compiler will decide which method to use based on the number and type of arguments you supply

Unsuccessful overloading

- Return type is NOT enough!!

```
int foo (double d);
double foo (double d);
```

- Won’t compile
- What if in my code, I just have

```
foo(3.0);
```

Overload example

```
class Overload{
        int r;
        String s;
        public void setValue (int r, String s) {
                this.r = r;        this.s = s;
        }
        public void setValue (String s, int r) {
                this.r =r;         this.s =s;
        }
        public static void main (String[] args){
                Overload o = new Overload();
                o.setValue(10, “ok”);
                o.setValue(“ok?”, 20); //both are OK!
        }
}
```

The compiler will decide which method to use based on the number and type of arguments you supply

Rewrite:

```
class Overload{
        int r;
        String s;
        public void setValue (int r, String s) {
                this.r = r;        this.s = s;
        }
        public void setValue (String s, int r) {
                this.setValue (r, s); //another usage of this
        }
        public static void main (String[] args){
                Overload o = new Overload();
                o.setValue(10, “ok”);
                o.setValue(“ok?”, 20); //both are OK!
        }
}
```

Avoid writing duplicate code

Multiple constructor

- Can invoke one constructor from another
- Use ***this(para)***
- Useful if constructors share a significant amount of initialization code, avoid repetition.
- Notice: this() must be the first statement in a constructor!! Can be called only once.

Example revisited

```
class Circle{
        double r;
        public Circle(){
                // r = 1.0; //default radius value;
                this (1.0); //call another constructor
        }
        public Circle (double r) {
                this.r = r;  //same name!
        }
        public static void main(String[] args){
                Circle c = new Circle(2.0); //OK
                Circle c2 = new Circle();    // OK now!
        }
}
```

How to initialize static fields?

- Cannot use constructor because no object created
- Static initializer:
  - **static** { *code to do initialization*}
  - Can appear anywhere in class definition where a field definition can.

```
public static String days = new String[7];
static{
		days[0]=“Monday”;
		days[1]=“Tuesday”;
		… …
}
```

Finalization – opposite of initialization

- Garbage collection can ONLY free the ***memory resources***
- Need finalize() to free other resources, for example, network connection, DB connection, file handler, etc.
- finalize() takes no argument, return void
- Invoked automatically by Java
- Rarely used for application-level programming

Reusing classes

- Suppose we want to define a class Student, which has name, weight, height, and school, gpa
- We can redefine everything
- Or, we can reuse the Person class since Student is one kind of Person, and just have additional attributes
    - Make Student inherit from Person
    - Avoid duplicate code

Inheritance

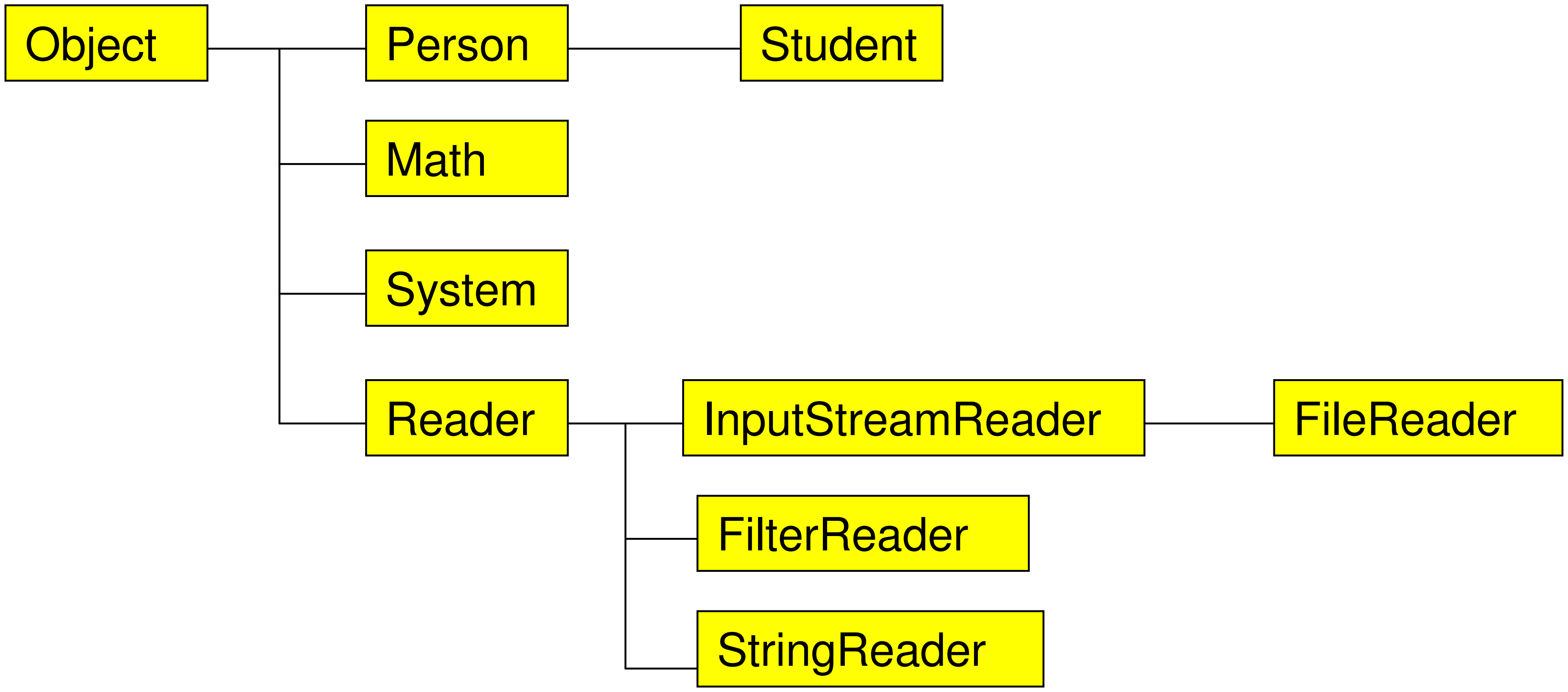
- Extends definition of existing class
- Keyword **extends**
  - Class Student extends Person{
- Subclass Student inherit the fields and methods from superclass Person

```
class Student extends Person{
            String school;
            double gpa;
}
```

Class Student automatically has fields name, weight, height, and all the methods defined in class Person

Object class

- If no superclass specified for a class, by default the superclass is java.lang.Object
  - All Java class inherit from Object
  - Object is the only one without a superclass



Upcasting

- The new class is a type of the existing class: Student is type of Person
- Any subclass object is also a legal superclass object, no casting needed. But the reverse need cast.
  - Student s=new Student();
  - Person p=s; //legal, auto upcasting
  - s=(B)p; //need specific cast, and only if p is pointing to a Student object
- p=s is legal, but p **cannot** use any extra fields/methods added by Student.

Example

```
class Hello{
        public static void main(String[] args){
                Student s = new Student();
                Person p = s; //legal
                p.school="columbia"; //Error!
        }
}
class Student extends Person{
        String school;
        double gpa;
}
```

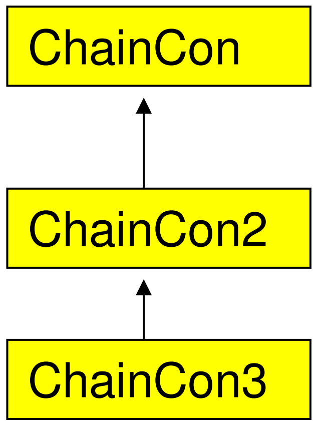
```
Hello.java:12: cannot resolve symbol
symbol  : variable school
location: class Person
                p.school="lala";
                 ^
1 error
```

Constructor chain

- In subclass constructor, use super(param) to invoke superclass constructor.
- Super() similar to this()
  - Used only in constructor
  - Must be the first statement
- By default, java implicitly call super() in subclass constructor, form the constructor chain.

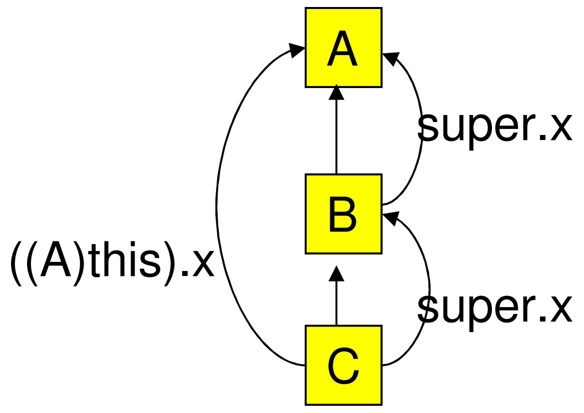
Example

- ChainCon.java



Shadowing superclass fields

- In subclass, use same field name, but different meaning
- To refer to the field in superclass, use keyword **super**, or type cast
  - Super can only refer to the direct parent, not grandparent



Each class has defined variable x, and use the following inside C:

```
x            // x in C, same as this.x
super.x    // x in B, same as ((B)this).x
super.super.x  // illegal, cannot point to A
((A)this).x // x in A
```

Override superclass method

- Define method with same name, return type and parameters in subclass
- When the method is invoked for an object of this class, the new definition of the method will be called, not the superclass one.
  - Runtime dynamic lookup to decide the type of object
- Overloading vs overriding
  - Overload: multiple definition for the same name method in the same class
  - Override: subclass re-define the same method from superclass
- Example: class A, B

Dynamic binding

- Binding: connecting a method call o a method body
- Early binding (static binding): binding is performed before the program is run
- Dynamic binding: binding occurs at run time, based on the type of the object
  - Java use this for all non-static methods
- Some type information stored in the object
- Example: shape, circle, square

final keyword

- If a class declared with the final modifier, then it cannot be extended or subclassed
- If a field is declared with final, then the value of it cannot be changed.
- If a method is declared with final, then it cannot be overridden in subclass

Access control modifier

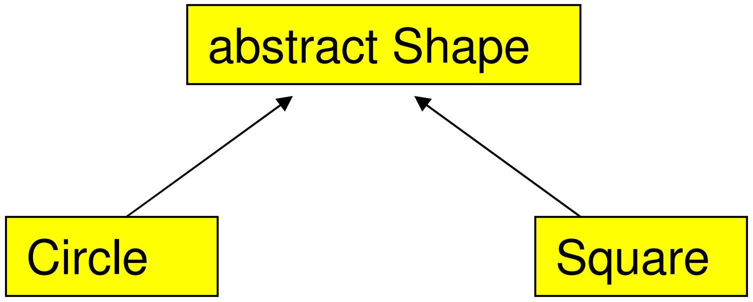
- Members of a class are always accessible within the body of the class
- **public**: accessible from outside of the class
- **private**: only within this class, even not visible in subclass
  - Subclass inherit it, but cannot directly access it inside the subclass
- **protected**: accessible from itself and all the subclasses

Abstract

- Sometimes it's helpful to have a common superclass, without any real implementation : abstract method
  - abstract return-type methodname(); //No {}!!
- A class with an abstract method must be declared as abstract also.
  - A class can be declared as abstract without having any abstract method
- An abstract class cannot be initiated
- Static, private, final methods cannot be abstract
- A subclass without implementing all the abstract class still be abstract

Example

- Revisit the Shape, Circle, Square code

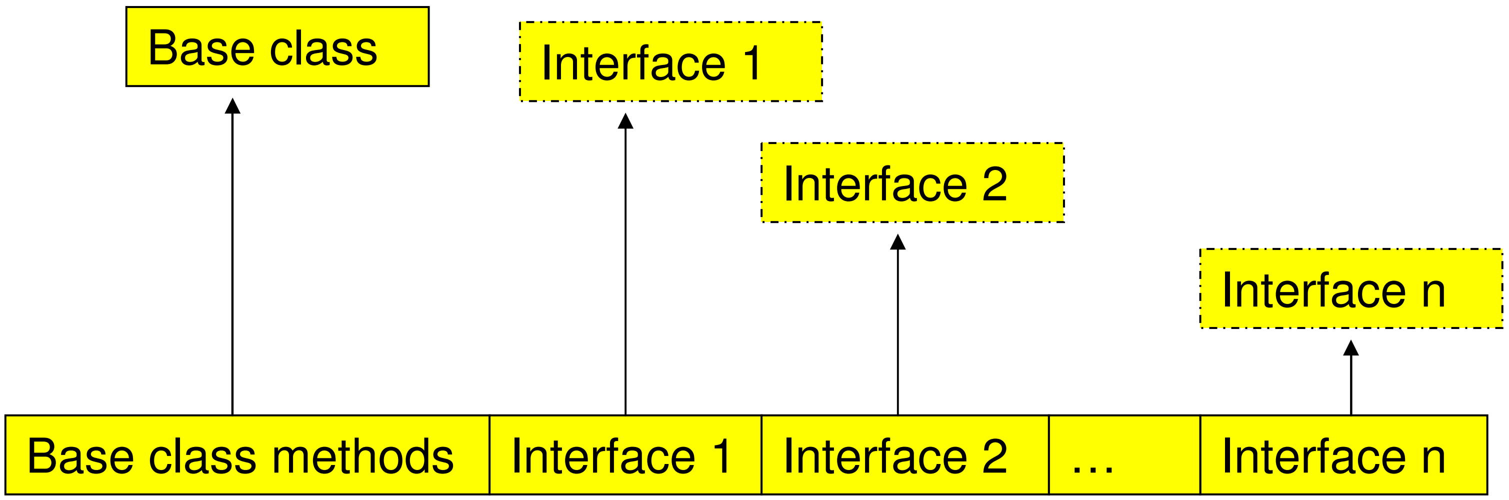


Interface

- A java class can extend only one superclass, but can implement multiple interface
- Interface is like a class, but it’s a *pure abstract* class
  - Define methods, but no implementation
  - Defines a public API. All methods are public.
    - No implementation, so nothing to hide
  - Cannot be instantiated, so no constructor

Implementing an interface

- Keyword: **implements**
- The class implementing the interface **has to** implement all the methods, otherwise declare as abstract



class A extends Base implements i1, i2, i3 { … }

Each interface becomes an independent type that you can upcast to.

Example

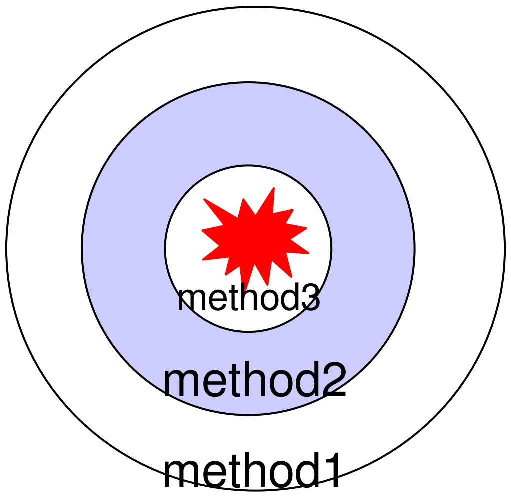
- Adventure.java

C++ features not in Java

- Multiple inheritance
  - interface kind of help on this one, since a class can implement multiple interface
- Template
- Operator overload

Explosions

- void method1() {…method2()}
- void method2() {…method3()}
- void method3() {…x=5/0} //BOOM!!



Error handling

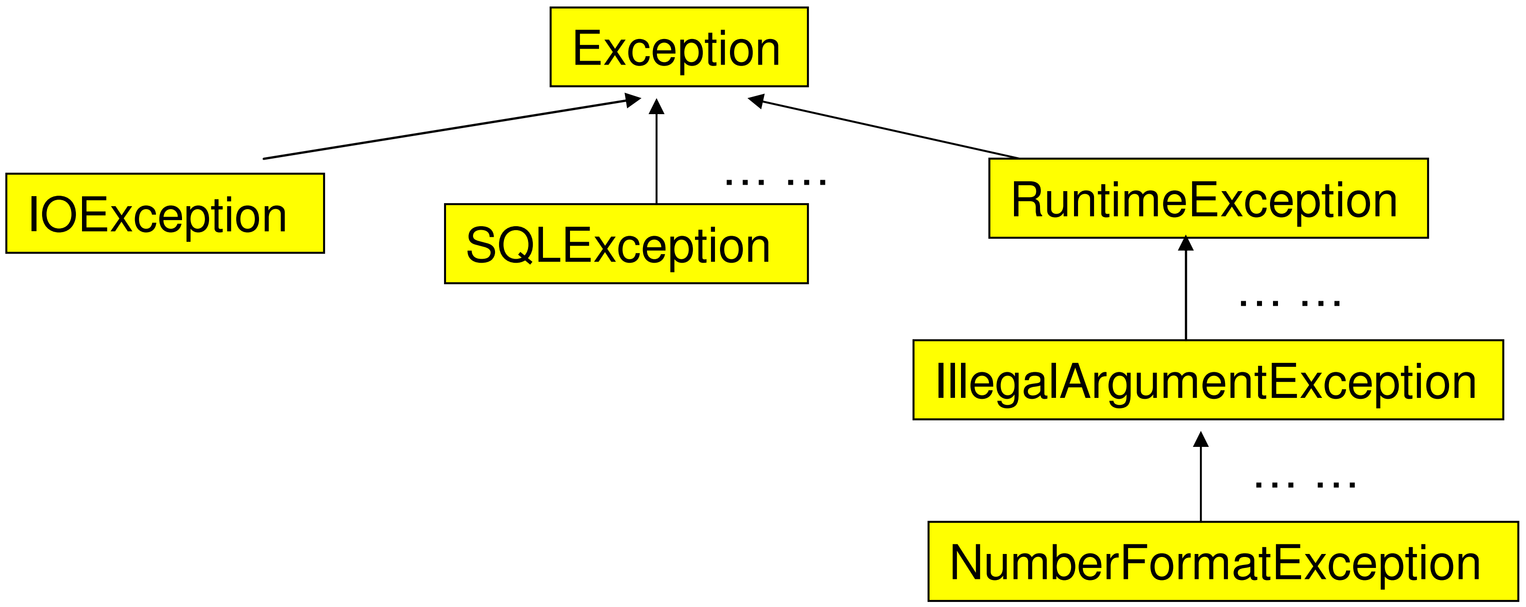
- Java philosophy: “badly formed code will not be run”
- Ideal time to catch error: compile
- Not all errors can be detected at compile time; the rest must be handled at run time
- Java: exception handling
  - The only official way that Java reports error
  - Enforced by compiler

Unexpected situation

- User input errors
- Device errors
- Physics limits
- Programmer errors

Exceptions are objects

- throw new IOException();
- throw new IOException(“file not open”);



Catching an exception

- Guarded region
  - Try block
  - Exception handler

```
try{
     //code that might generate exceptions
} catch (Type1 id1){
    // handle exception for Type1
} catch (Type2 id2){
    // handle exception for Type2
}
```

Only the first catch block with matching exception type will be execute

Create your own exception

- Create your own to denote a special problem
- Example: ExceptionTest.java

RuntimeException

- Always thrown automatically by Java
- You can **only** ignore RuntimeException in coding, all other handling is carefully enforced by compiler
  - RuntimeException represents programming error
    - NullPointerException
    - ArrayIndexOutOfBoundsException
    - NumberFormatException

Finally clause – clean up

- Always execute, regardless of whether the body terminates normally or via exception
- Provides a good place for required cleanup
  - Generally involves releasing resources, for example, close files or connections

```
try{
     //code that might throw A or B exception
} catch (A a){
    // handler for A
} catch (B b){
    //handler for B
} finally {
    //activities that happen every time, do cleanup
}
```

When to use Exception

- 90% of time: because the Java libraries force you to
- Other 10% of the time: your judgement
- Software engineering rule of thumb
  - Your method has *preconditions* and *postcondition*
  - If preconditions are met, but you can't fulfill your postconditions, throw an exception