

Object-Oriented Programming using Java

Different Programming Paradigms

- Functional/procedural programming:
 - program is a list of instructions to the computer
- Object-oriented programming
 - program is composed of a collection *objects* that communicate with each other

Main Concepts

- Object
- Class
- Inheritance
- Encapsulation

Objects

- identity – unique identification of an object
- attributes – data/state
- services – methods/operations
 - supported by the object
 - within objects responsibility to provide these services to other clients

Class

- “type”
- object is an **instance** of class
- class groups similar objects
 - same (structure of) attributes
 - same services
- object holds values of its class’s attributes

Inheritance

- Class hierarchy
- Generalization and Specialization
 - subclass inherits attributes and services from its superclass
 - subclass may add new attributes and services
 - subclass may reuse the code in the superclass
 - subclasses provide specialized behaviors (overriding and dynamic binding)
 - partially define and implement common behaviors (abstract)

Encapsulation

- Separation between internal state of the object and its external aspects
- How ?
 - control access to members of the class
 - **interface** “type”

What does it buy us ?

- Modularity
 - source code for an object can be written and maintained independently of the source code for other objects
 - easier maintenance and reuse
- Information hiding
 - other objects can ignore implementation details
 - security (object has control over its internal state)
- but
 - shared data need special design patterns (e.g., DB)
 - performance overhead

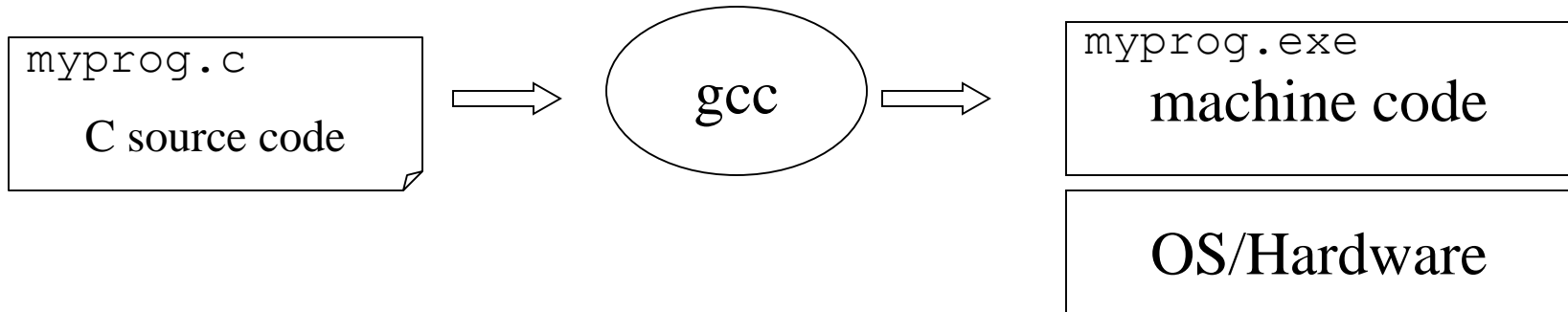
Why Java ?

- Portable
- Easy to learn
- [Designed to be used on the Internet]

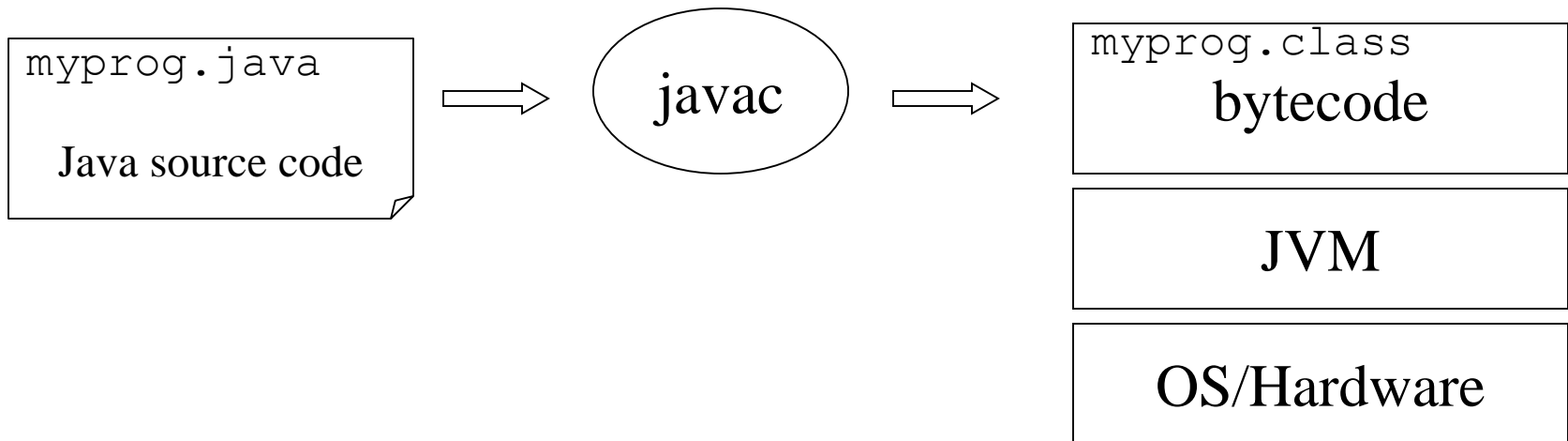
JVM

- JVM stands for
Java Virtual Machine
- Unlike other languages, Java “executables” are executed on a CPU that does not exist.

Platform Dependent



Platform Independent



Primitive types

- int 4 bytes
- short 2 bytes
- long 8 bytes
- byte 1 byte
- float 4 bytes
- double 8 bytes
- char Unicode encoding (2 bytes)
- boolean {true,false}

*Behaviors is
exactly as in
C++*

Note:
*Primitive type
always begin
with lower-case*

Java Tokens

Smallest individual unit in source code.

- Keywords
- Identifiers
- Literals
- Operators
- Separators

Java Data types

- Data types in Java are classified into two types:
 - Primitive—which include Integer, Character, Boolean, and Floating Point.
 - Non-primitive—which include Classes, Interfaces, and Arrays.
- **Primitive Data Types**
 - **1. Integer**
 - Integer types can hold whole numbers such as 123 and -96. The size of the values that can be stored depends on the integer type that we choose.
 - **2. Floating Point**
 - Floating point data types are used to represent numbers with a fractional part. Single precision floating point numbers occupy 4 bytes and Double precision floating point numbers occupy 8 bytes. There are two subtypes:
 - **3. Character**
 - It stores character constants in the memory. It assumes a size of 2 bytes, but basically it can hold only a single character because char stores unicode character sets. It has a minimum value of 'u0000' (or 0) and a maximum value of 'uffff' (or 65,535, inclusive).
 - **4. Boolean**
 - Boolean data types are used to store values with two states: true or false.

Java's "Building Blocks"

- Java programs relies on software components called objects
 - An object contains both data and behavior
 - An object is defined by a class
- A program is made up of one or more classes
 - A class contains one or more methods
 - A method contains program statements
- Statements are
 - Variable declarations: primitive data types and classes
 - Operations: arithmetic, logical, bit-level, class access
 - Control structures: selection, looping, etc.
 - Object messages: i.e., calls to methods
- A Java application always executes the main method

Primitive types - cont.

- **Constants**

37 integer

37.2 float

42F float

0754 integer (octal)

0xfe integer (hexadecimal)

Java Versus C++

- Java is a true object oriented language while C++ is basically C with an object oriented extension. Listed below are some major C++ features that were intentionally omitted from Java, or significantly modified:
- Java does not support operator overloading.
- Java does not have template classes.
- Java does not support multiple inheritance of classes.
- Java does not support global variables.
- Java does not use pointers.
- Java has replaced the destructor function with the `finalize()` function.
- There are no header files in Java.

Wrappers

Java provides Objects which wrap primitive types and supply methods.

Example:

```
Integer n = new Integer("4");  
int m = n.intValue();
```

[Read more about Integer in JDK Documentation](#)

More sophisticated

```
class Kyle {  
    private boolean kennyIsAlive_  
    public Kyle() { kennyIsAlive_ = true; }  
    public Kyle(Kyle aKyle) {  
        kennyIsAlive_ = aKyle.kennyIsAlive_  
    }  
    public String theyKilledKenny() {  
        if (kennyIsAlive_) {  
            kennyIsAlive_ = false;  
            return "You bastards !!!";  
        } else {  
            return "?";  
        }  
    }  
    public static void main(String[] args) {  
        Kyle k = new Kyle();  
        String s = k.theyKilledKenny();  
        System.out.println("Kyle: " + s);  
    }  
}
```

Default
C'tor

Copy
C'tor

Results

```
javac Kyle.java      ( to compile )
```

```
java Kyle            ( to execute )
```

```
Kyle: You bastards !!!
```

Arrays

- Array is an object
- Array size is fixed

```
Animal[] arr; // nothing yet ...
```

```
arr = new Animal[4]; // only array of pointers
```

```
for(int i=0 ; i < arr.length ; i++) {  
    arr[i] = new Animal();  
}
```

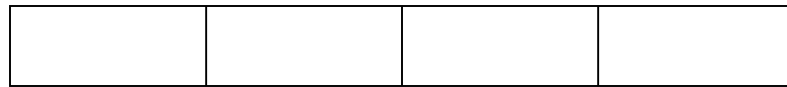
```
// now we have a complete array
```

Arrays - Multidimensional

- In C++

```
Animal arr[2][2]
```

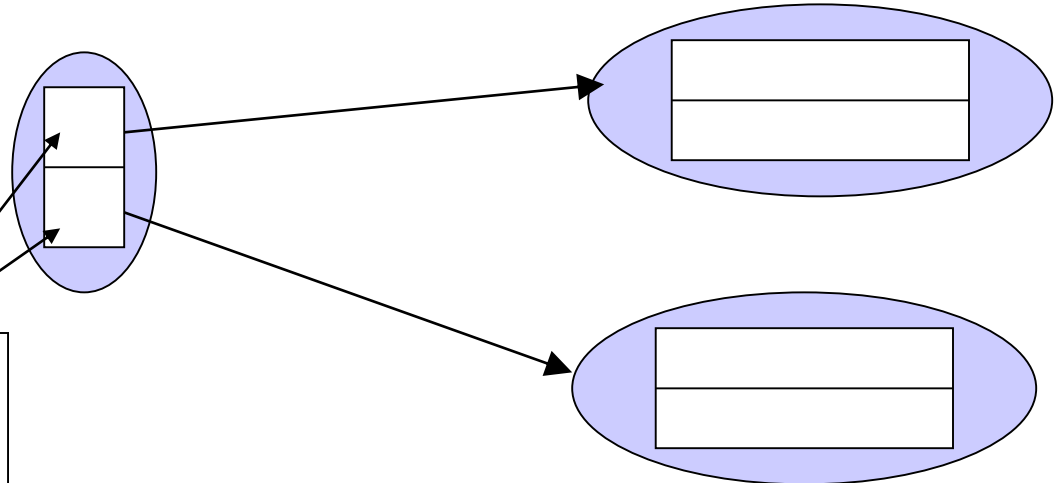
Is:



-
- In Java

```
Animal[][] arr=  
new Animal[2][2]
```

What is the type of
the object here ?



Static - [1/4]

- **Member data** - Same data is used for all the instances (objects) of some Class.

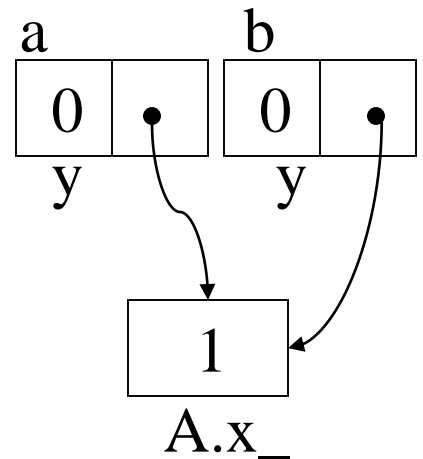
```
Class A {  
    public int y = 0;  
    public static int x_ = 1;  
};
```

*Assignment performed on the first access to the Class.
Only one instance of 'x' exists in memory*

```
A a = new A();  
A b = new A();  
System.out.println(b.x_);  
a.x_ = 5;  
System.out.println(b.x_);  
A.x_ = 10;  
System.out.println(b.x_);
```

Output:

1
5
10



Static - [2/4]

- **Member function**

- Static member function can access only static members
- Static member function can be called without an instance.

```
Class TeaPot {
    private static int numOfTP = 0;
    private Color myColor_;
    public TeaPot(Color c) {
        myColor_ = c;
        numOfTP++;
    }
    public static int howManyTeaPots()
        { return numOfTP; }

    // error :
    public static Color getColor()
        { return myColor_; }
}
```

Static - [2/4] cont.

Usage :

```
TeaPot tp1 = new TeaPot(Color.RED);
```

```
TeaPot tp2 = new TeaPot(Color.GREEN);
```

```
System.out.println("We have " +  
    TeaPot.howManyTeaPots() + "Tea Pots");
```

Static - [3/4]

- **Block**

- Code that is executed in the first reference to the class.
- Several static blocks can exist in the same class
(Execution order is by the appearance order in the class definition).
- Only static members can be accessed.

```
class RandomGenerator {
    private static int seed_;

    static {
        int t = System.getTime() % 100;
        seed_ = System.getTime();
        while(t-- > 0)
            seed_ = getNextNumber(seed_);
    }
}
```

String is an Object

- Constant strings as in C, does not exist
- The function call `foo("Hello")` creates a String object, containing “Hello”, and passes reference to it to `foo`.
- There is no point in writing :

```
String s = new String("Hello");
```

- The String object is a constant. It can't be changed using a reference to it.

Flow control

Basically, it is exactly like c/c++.

if/else

```
If (x==4) {  
    // act1  
} else {  
    // act2  
}
```

do/while

```
int i=5;  
do {  
    // act1  
    i--;  
} while (i!=0);
```

for

```
int j;  
for (int i=0; i<=9; i++)  
{  
    j+=i;  
}
```

switch

```
char  
c=IN.getChar();  
switch (c) {  
    case 'a':  
        // act1  
        break;  
    default:  
        // act2  
}
```

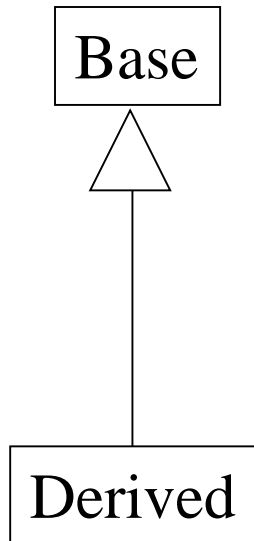
Packages

- Java code has hierarchical structure.
- The environment variable `CLASSPATH` contains the directory names of the roots.
- Every Object belongs to a package (‘package’ keyword)
- Object full name contains the name full name of the package containing it.

Access Control

- *public* member (function/data)
 - Can be called/modified from outside.
- *protected*
 - Can be called/modified from derived classes
- *private*
 - Can be called/modified only from the current class
- *default (if no access modifier stated)*
 - Usually referred to as “Friendly”.
 - Can be called/modified/instantiated from the same package.

Inheritance



```
class Base {
    Base() {}
    Base(int i) {}
    protected void foo() {...}
}

class Derived extends Base {
    Derived() {}
    protected void foo() {...}
    Derived(int i) {
        super(i);
        ...
        super.foo();
    }
}
```

As opposed to C++, it is possible to inherit only from ONE class.

Pros avoids many potential problems and bugs.

Cons might cause code replication

Polymorphism

- Inheritance creates an “is a” relation:

For example, if B inherits from A, than we say that “B is also an A”.

Implications are:

- access rights (Java forbids reducing access rights) - derived class can receive all the messages that the base class can.
- behavior
- precondition and postcondition

Inheritance (2)

- In Java, all methods are virtual :

```
class Base {
    void foo() {
        System.out.println("Base");
    }
}
class Derived extends Base {
    void foo() {
        System.out.println("Derived");
    }
}
public class Test {
    public static void main(String[] args) {
        Base b = new Derived();
        b.foo(); // Derived.foo() will be activated
    }
}
```

Inheritance (3) - Optional

```
class classC extends classB {
    classC(int arg1, int arg2){
        this(arg1);
        System.out.println("In classC(int arg1, int arg2)");
    }
    classC(int arg1){
        super(arg1);
        System.out.println("In classC(int arg1)");
    }
}
class classB extends classA {
    classB(int arg1){
        super(arg1);
        System.out.println("In classB(int arg1)");
    }
    classB(){
        System.out.println("In classB()");
    }
}
```

Inheritance (3) - Optional

```
class classA {
    classA(int arg1){
        System.out.println("In classA(int arg1)");
    }
    classA(){
        System.out.println("In classA()");
    }
}

class classB extends classA {
    classB(int arg1, int arg2){
        this(arg1);
        System.out.println("In classB(int arg1, int arg2)");
    }
    classB(int arg1){
        super(arg1);
        System.out.println("In classB(int arg1)");
    }
}

class B() {
    System.out.println("In classB()");
}
}
```

Abstract

- *abstract* member function, means that the function does not have an implementation.
- *abstract* class, is class that can not be instantiated.

```
AbstractTest.java:6: class AbstractTest is an abstract class.  
It can't be instantiated.
```

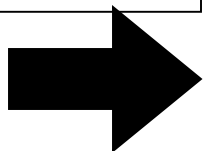
```
    new AbstractTest();  
    ^
```

```
1 error
```

NOTE:

An abstract class is **not** required to have an abstract method in it. But any class that has an abstract method in it or that does not provide an implementation for any abstract methods declared in its superclasses **must** be declared as an abstract class.

Example



Abstract - Example

```
package java.lang;
public abstract class Shape {
    public abstract void draw();
    public void move(int x, int y) {
        setColor(BackgroundColor);
        draw();
        setCenter(x,y);
        setColor(ForegroundColor);
        draw();
    }
}
```

```
package java.lang;
public class Circle extends Shape {
    public void draw() {
        // draw the circle ...
    }
}
```

Interface

Interfaces are useful for the following:

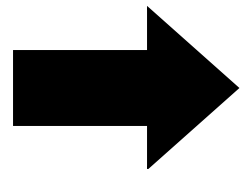
- Capturing similarities among unrelated classes without artificially forcing a class relationship.
- Declaring methods that one or more classes are expected to implement.
- Revealing an object's programming interface without revealing its class.

Interface

- abstract “class”
- Helps defining a “usage contract” between classes
- All methods are public
- Java’s compensation for removing the multiple inheritance. You can “inherit” as many interfaces as you want.

* - The correct term is “to implement”
an interface

Example



Interface

```
interface IChef {  
    void cook(Food food);  
}
```

```
interface BabyKicker {  
    void kickTheBaby(Baby);  
}
```

```
interface SouthParkCharacter {  
    void curse();  
}
```

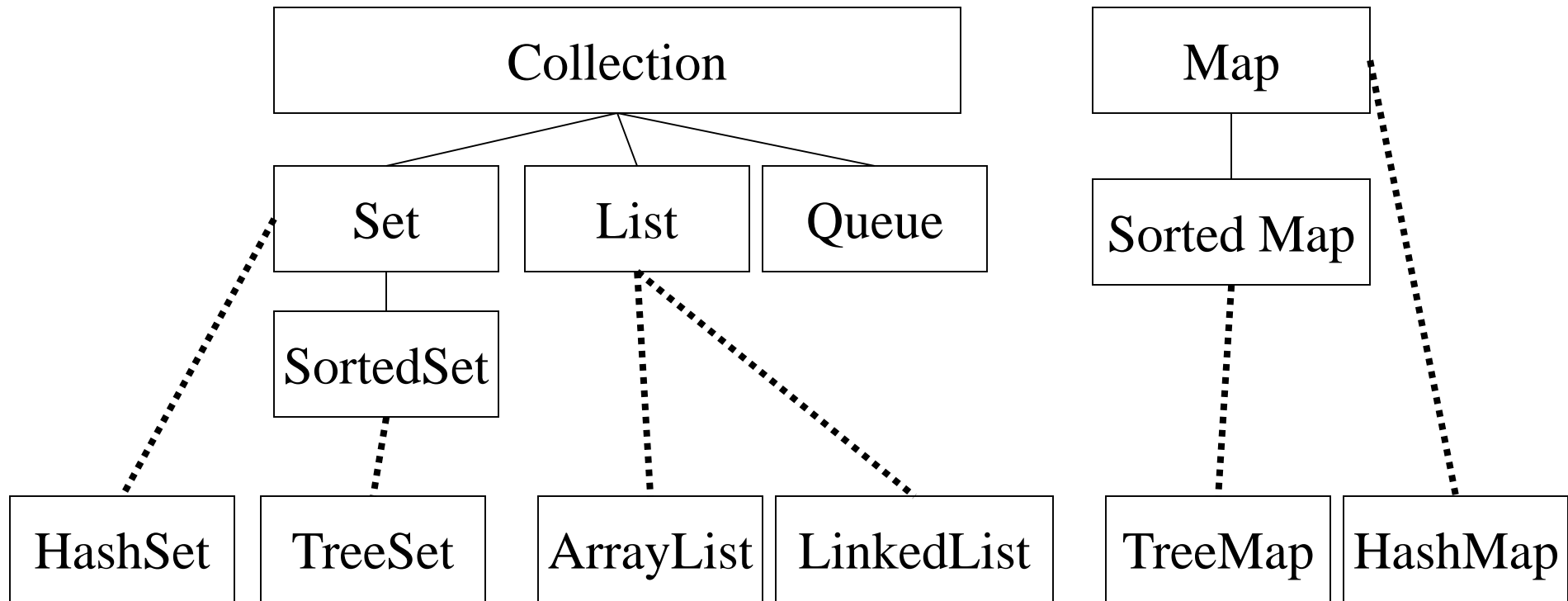
```
class Chef implements IChef, SouthParkCharacter {  
    // overridden methods MUST be public  
    // can you tell why ?  
    public void curse() { ... }  
    public void cook(Food f) { ... }  
}
```

* access rights (Java forbids reducing of access rights)

When to use an interface ?

Perfect tool for encapsulating the classes inner structure. Only the interface will be exposed

General Purpose Implementations



```
List<String> list1 = new ArrayList<String>(c);  
List<String> list2 = new LinkedList<String>(c);
```

final

- ***final* member data**

Constant member

- ***final* member function**

The method can't be overridden.

- ***final* class**

'Base' is final, thus it can't be extended

(String class is final)

```
final class Base {
    final int i=5;
    final void foo() {
        i=10;
        //what will the compiler say
        //about this?
    }
}

class Derived extends Base {
    // Error
    // another foo ...
    void foo() {

    }
}
```

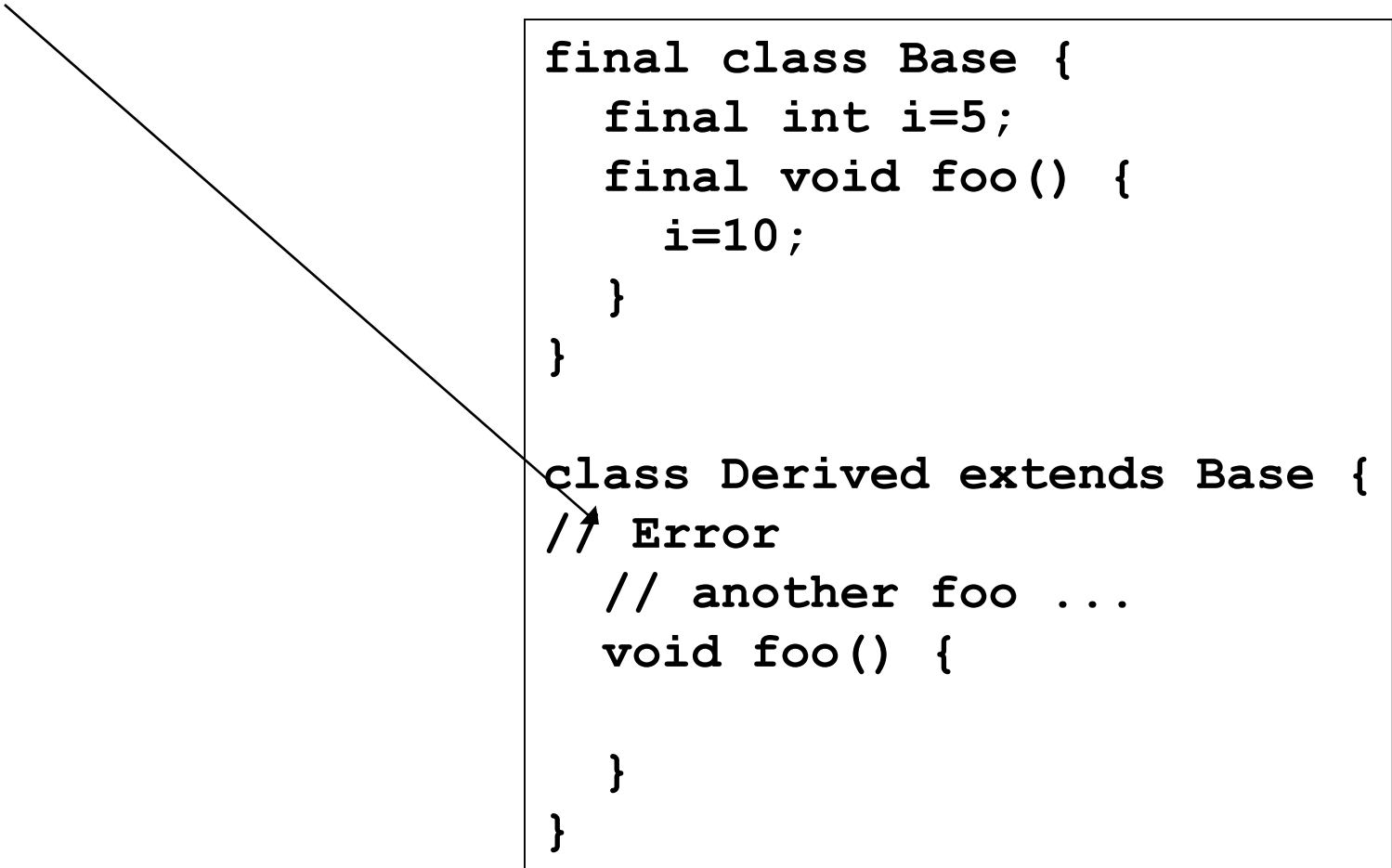
final

```
Derived.java:6: Can't subclass final classes: class Base  
class class Derived extends Base {
```

^

1 error

```
final class Base {  
    final int i=5;  
    final void foo() {  
        i=10;  
    }  
}  
  
class Derived extends Base {  
    // Error  
    // another foo ...  
    void foo() {  
  
    }  
}
```



Exception - What is it and why do I care?

Definition: An *exception* is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

- Exception is an Object
- Exception class must be descendent of Throwable.

Exception - What is it and why do I care?(2)

By using exceptions to manage errors, Java programs have the following advantages over traditional error management techniques:

- 1: Separating Error Handling Code from "Regular" Code**
- 2: Propagating Errors Up the Call Stack**
- 3: Grouping Error Types and Error Differentiation**

1: Separating Error Handling Code from "Regular" Code (1)

```
readFile {  
    open the file;  
    determine its size;  
    allocate that much memory;  
    read the file into memory;  
    close the file;  
}
```


1: Separating Error Handling Code from "Regular" Code (2)

```
errorCodeType readFile {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDintClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

1: Separating Error Handling Code from "Regular" Code (3)

```
readFile {  
    try {  
        open the file;  
        determine its size;  
        allocate that much memory;  
        read the file into memory;  
        close the file;  
    } catch (fileOpenFailed) {  
        doSomething;  
    } catch (sizeDeterminationFailed) {  
        doSomething;  
    } catch (memoryAllocationFailed) {  
        doSomething;  
    } catch (readFailed) {  
        doSomething;  
    } catch (fileCloseFailed) {  
        doSomething;  
    }  
}
```

2: Propagating Errors Up the Call Stack

```
method1 {  
    try {  
        call method2;  
    } catch (exception) {  
        doErrorProcessing;  
    }  
}  
method2 throws exception {  
    call method3;  
}  
method3 throws exception {  
    call readFile;  
}
```