# Lecture1:Introduction To Java and Object-Oriented Paradigm

EC3375:Software Design Pattern and Technology

# 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**

# 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 maintainance 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

# JAVA

*mainly for c++ programmer*

# Why Java ?

- Portable
- Easy to learn

- [ Designed to be used on the Internet ]

# What is Java?

- Object-oriented programming (OOP) language developed by SUN Microsystems
- Similar to C and C++, except without some of the confusing, poorly understood features of C++
- Extensive networking facilities
- Extensive set APIs for GUIs, distributed computing, 2d/3D graphics and others

# The Java programming environment

- Compared to C++:
  - no header files, macros, pointers and references, unions, operator overloading, templates, etc.
- *Object-orientation*: Classes + Inheritance
- *Distributed*: RMI, Servlet, Distributed object programming.
- *Robust*: Strong typing + no pointer + garbage collection
- *Secure*: Type-safety + access control

# The Java programming environment (Cont..)

- *Portable*
- *Interpreted*
  - *High performance through* Just in time compilation + runtime modification of code
- *Multi-threaded*

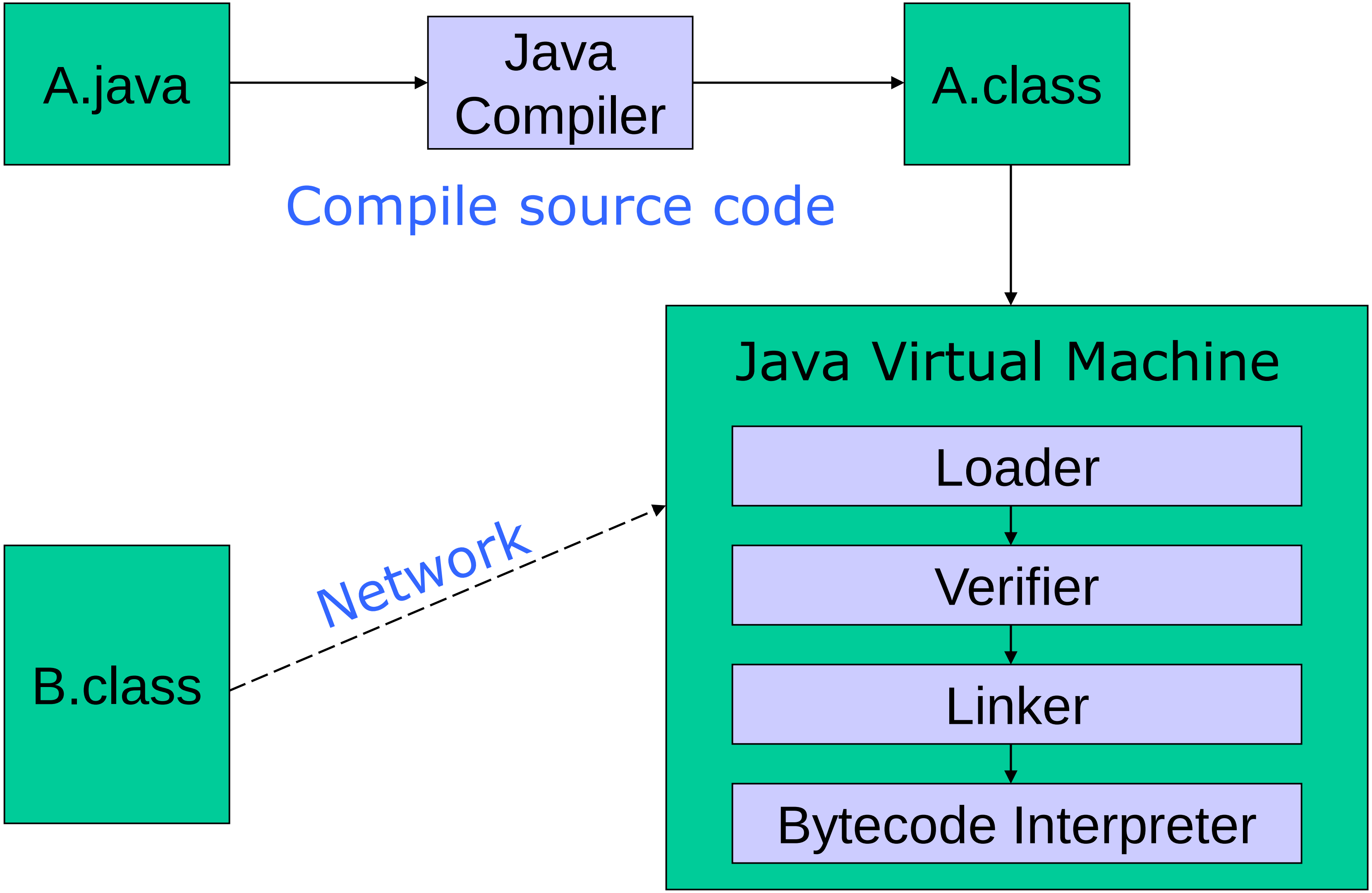
# JVM

- JVM stands for

  **J**ava **V**irtual **M**achine

- Unlike other languages, Java “executables” are executed on a CPU that does not exist.

# Java Virtual Machine Architecture



# Java Virtual Machine

- Java is compiled into bytecodes
- Bytecodes are high-level, machine-independent instructions.
- The Java run-time system provides the JVM
- The JVM interprets the bytecodes during program execution.

# Types Of Java Programs

- Application
  - Standalone Java program that can run independent of any Web browser
- Applet
  - Java program that runs within a Java-enabled Web browser
- Servlet
  - Java software that is loaded a Web server to provide additional server functionality ala CGI programs

# 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} |

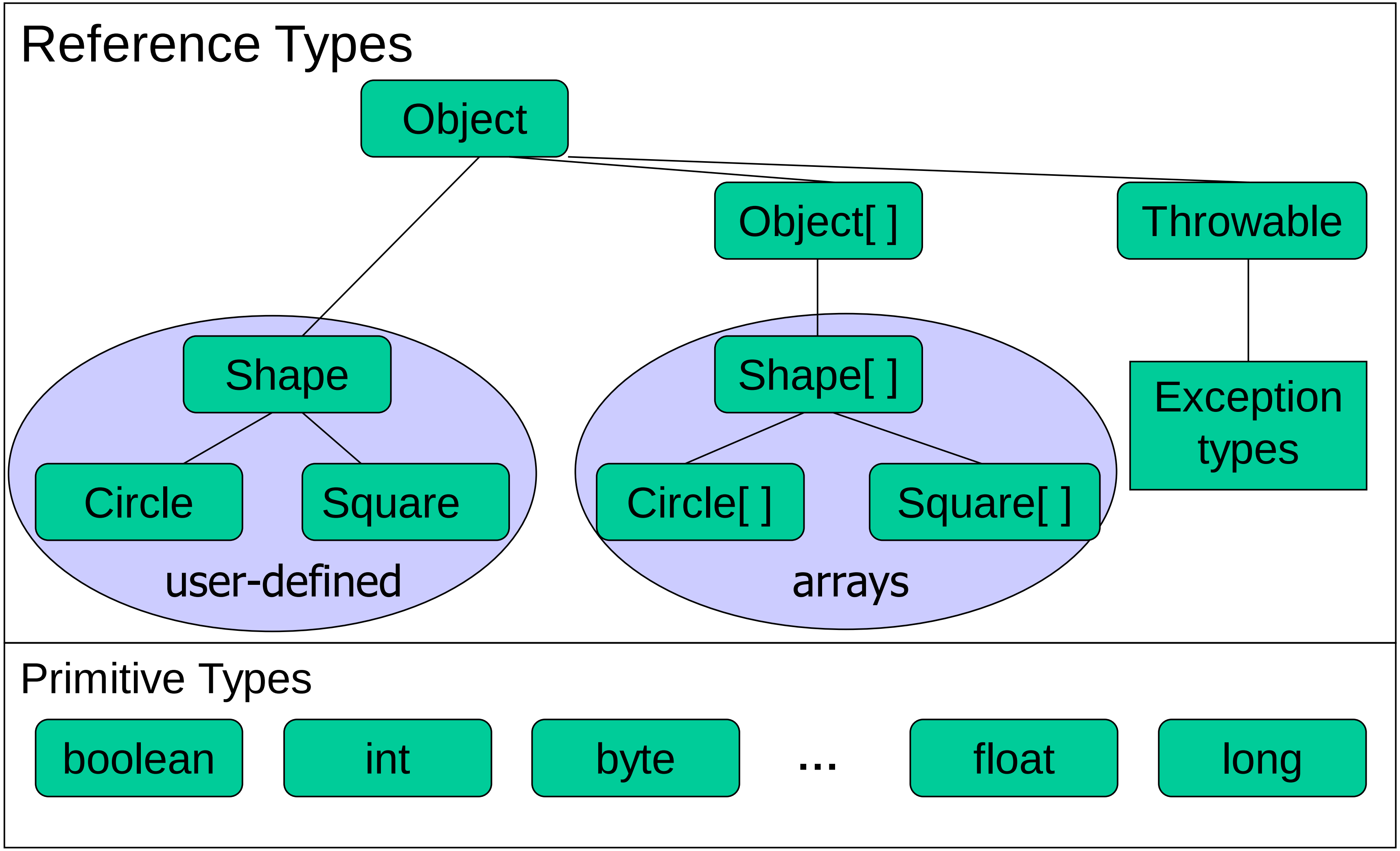
***Note:***
*Primitive type always begin with lower-case*

# Primitive types - cont.

- **Constants**

| | |
|---|---|
| 37 | integer |
| 37.2 | float |
| 42F | float |
| 0754 | integer (octal) |
| 0xfe | integer (hexadecimal) |

# Classification of Java types



# 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

# Hello World

Hello.java

```
class Hello {
      public static void main(String[] args) {
            System.out.println(“Hello World !!!”);
      }
}
```

C:\javac Hello.java *( compilation creates Hello.class )*

C:\java Hello *(Execution on the local JVM)*

# More sophisticated

Default C’tor

Copy C’tor

```
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 !!!”;
        } else {
            return “?”;
        }
    }
    public static void main(String[] args) {
        Kyle k = new Kyle();
        String s = k.theyKilledKenny();
        System.out.println(“Kyle: “ + s);
    }
}
```

# Results

```
javac Kyle.java       ( to compile )

java Kyle             ( to execute )

Kyle:  !!!
```

# Arrays

- Array is an object
- Array size is fixed

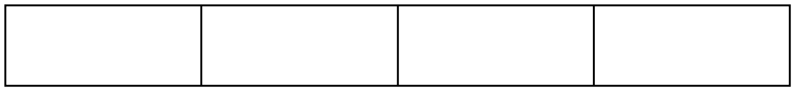
```
Animal[] arr; // nothing yet …

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++

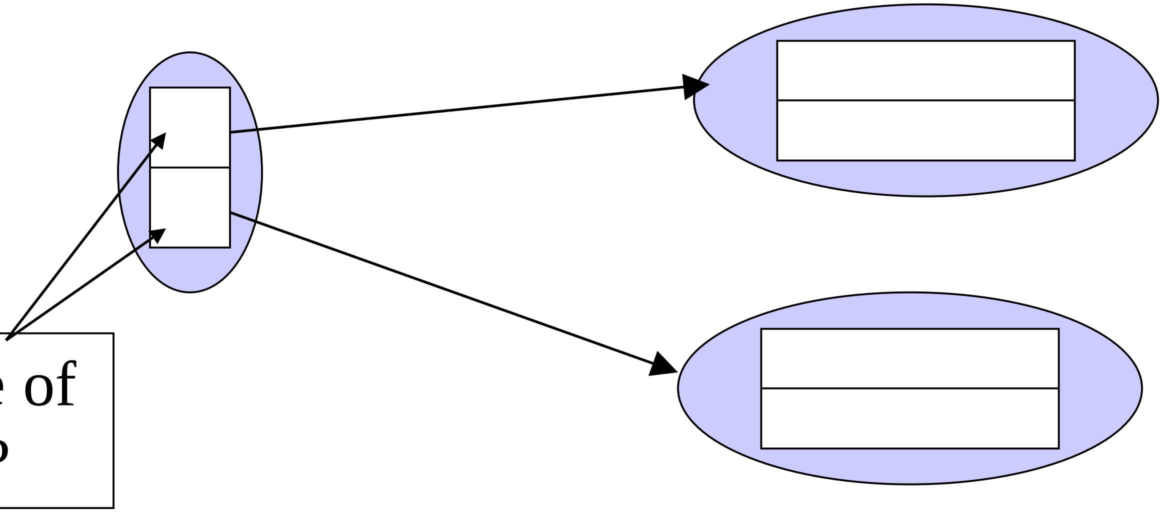
```
Animal arr[2][2]
```

Is:



- In Java

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



What is the type of the object here ?

# Static - [1/4]

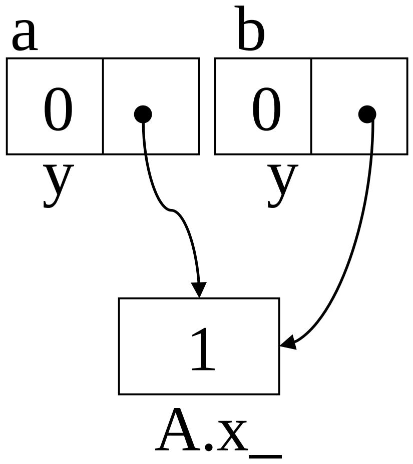
- **<u>Member data</u>** - Same data is used for all the instances (objects) of some Class.

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

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_);
```

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

**<u>Output</u>:**

```
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’:
  case ‘b’:
    // act1
    break;
  default:
    // act2
}
```

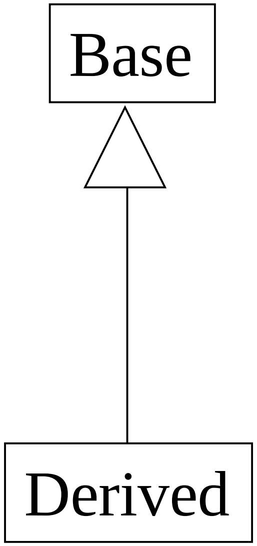
# Packages

- 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.

Example

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

# 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

# Collections

- Collection/container
  - – object that groups multiple elements
  - – used to store, retrieve, manipulate, communicate aggregate data
- Iterator - object used for traversing a collection and selectively remove elements
- Generics – implementation is parametric in the type of elements

# Java Collection Framework

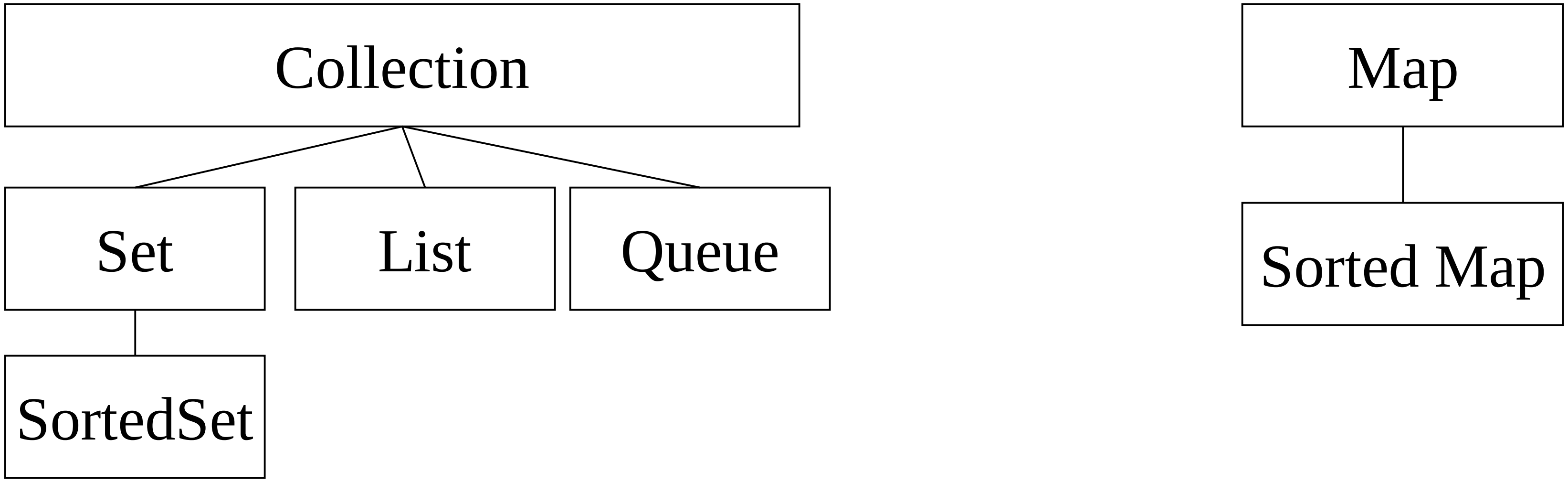
- Goal: Implement reusable data-structures and functionality

- Collection interfaces - manipulate collections independently of representation details
- Collection implementations - reusable data structures

```
List<String> list = new ArrayList<String>(c);
```

- Algorithms - reusable functionality
  - computations on objects that implement collection interfaces
  - e.g., searching, sorting
  - polymorphic: the same method can be used on many different implementations of the appropriate collection interface

# Collection Interfaces



Collection

Set

List

Queue

SortedSet

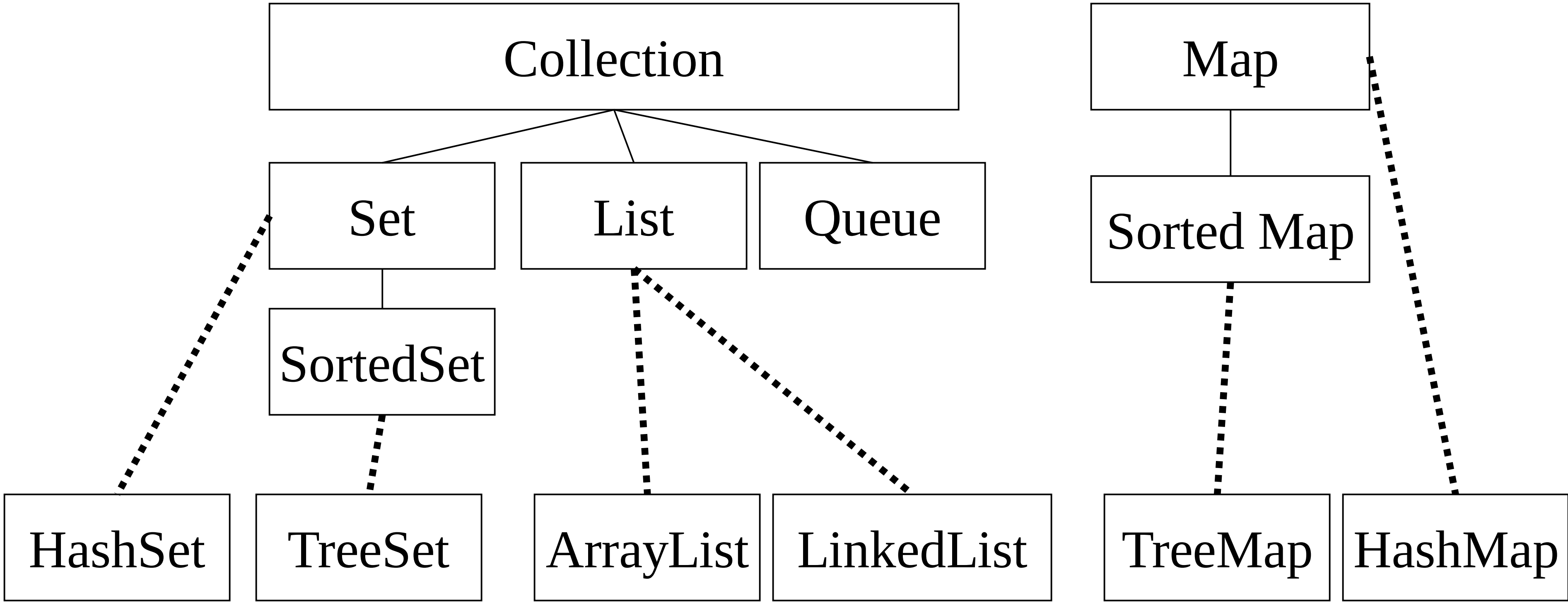
Map

Sorted Map

# Collection Interface

- Basic Operations
  - int size();
  - boolean isEmpty();
  - boolean contains(Object element);
  - boolean add(E element);
  - boolean remove(Object element);
  - Iterator iterator();
- Bulk Operations
  - boolean containsAll(Collection<?> c);
  - boolean addAll(Collection<? extends E> c);
  - boolean removeAll(Collection<?> c);
  - boolean retainAll(Collection<?> c);
  - void clear();
- Array Operations
  - Object[] toArray(); <T> T[] toArray(T[] a); }

# 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() {

  }
}
```

# IO - Introduction

- Definition
  - **Stream** is a flow of data
    - characters read from a file
    - bytes written to the network
    - …

- Philosophy
  - All streams in the world are basically the same.
  - Streams can be divided (as the name “IO” suggests) to **Input** and **Output** streams.

- Implementation
  - Incoming flow of data (characters) implements “Reader” (InputStream for bytes)
  - Outgoing flow of data (characters) implements “Writer” (OutputStream for bytes –eg. Images, sounds etc.)

# 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 (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

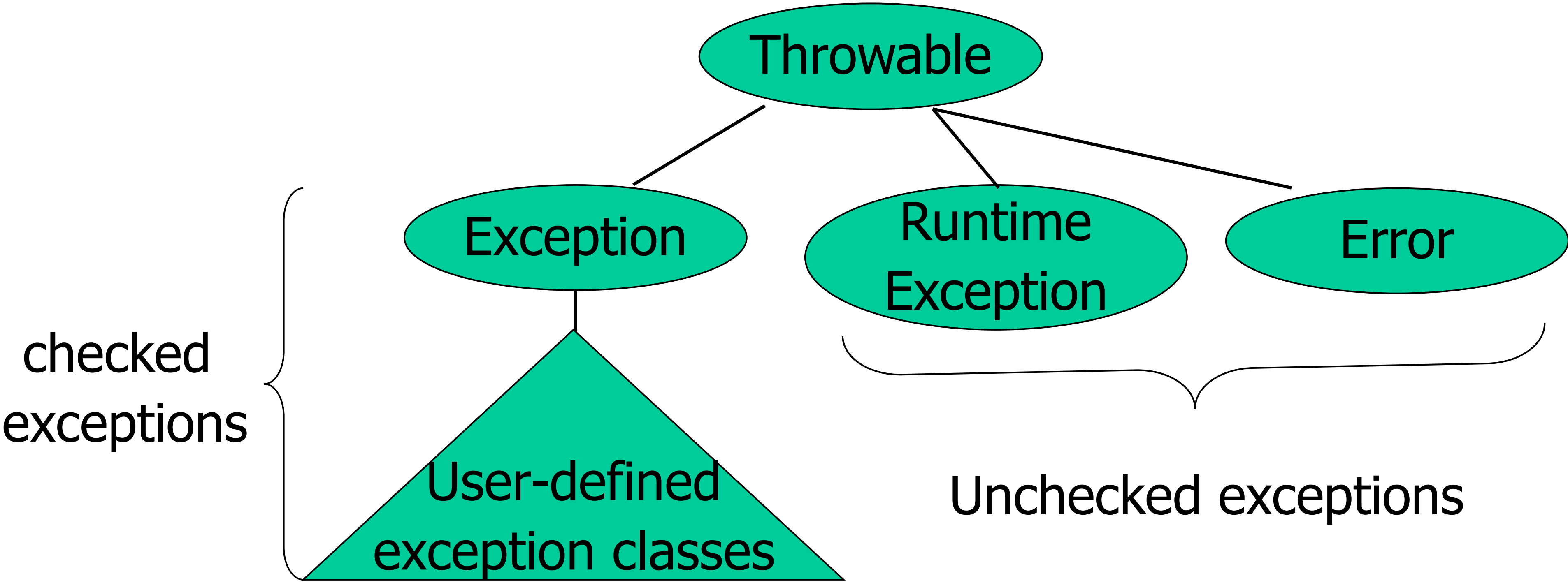
# 1: Separating Error Handling Code from "Regular" Code (3)

```
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;
}
```

# Exception Classes



- If a method may throw a checked exception, then this must be in the type of the method

# Exception Classes

- **Checked exceptions vs. unchecked exceptions.**
  - Compiler enforces a catch-or-declare requirement for checked exceptions.
- An exception's type determines whether it is checked or unchecked.
- Direct or indirect subclasses of class RuntimeException (package `java.lang`) are *unchecked* exceptions.
  - Typically caused by defects in your program's code (e.g., `ArrayIndexOutOfBoundsException`s).
- Subclasses of `Exception` but not `RuntimeException` are *checked* exceptions.
  - Caused by conditions that are not in the control of the program—e.g., in file processing, the program can't open a file because the file does not exist.