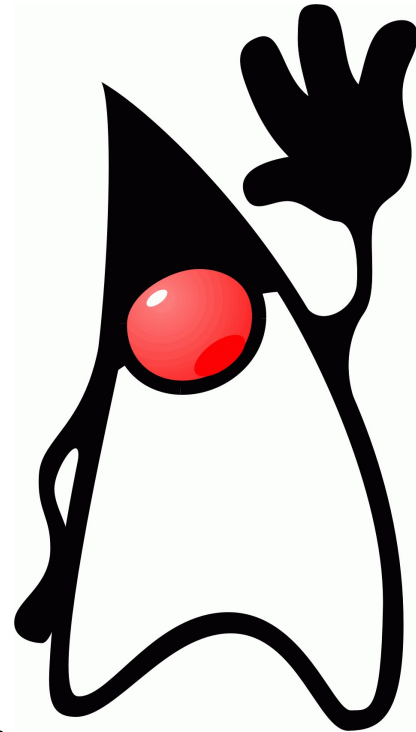


# MSc Introduction to Java

---



**Dr Maria Valera Espina**  
**Department of Computing**  
**Imperial College London**  
**January 2015**

Based on William Knottenbelt , Reuben Owen and Alastair Donalson

# Course Overview

---

- Lectures:
  - Tuesday 13<sup>th</sup> January 4-6pm
    - The Java Framework (An Overview)
    - Language specification (A more detailed look at the language)
    - Input/Output
  - Friday 16<sup>th</sup> January 10am-12pm
    - Casting objects
    - Generics, containers and collections
    - Exceptions
  - Friday 16<sup>th</sup> January 2-3pm
    - Graphical User Interface (GUI), Threads and Basic Networking
- Labs:
  - Thursday 16<sup>th</sup> January 4-6pm
  - Friday 17<sup>th</sup> January 3-6pm

## Reference Materials

---


- **The Java Tutorial** online at:  
`http://docs.oracle.com/javase/tutorial/`
- **Java API Documentation**  
`http://docs.oracle.com/javase/7/docs/`
- **Java – The Complete Reference (8th ed.)** (McGraw-Hill)  
by Herbert Schildt
- **Developing Java Software** (Wiley)  
by Russel Winder and Graham Roberts
- **Thinking in Java** (Prentice-Hall)  
by Bruce Eckel, also online:  
`http://www.mindview.net/Books/TIJ/`

# What is an OO Programming Language?

---

- Essentially OO Program does:
  - Creates objects
  - Requests that objects perform services by sending them **messages** (i.e. calling methods)
- Once an object receives a message takes control of execution.  
It may:
  - Update its state
  - Update state of others objects
  - Send message to other objects
- Once an object has dealt with a message, it returns control to the object that sent the message

In general, bad practice



Usually best practice



# Which OOP?

---

- Java. Why?
  - It is a pure OOP
  - Widespread in Industry
    - ♦ Traditional IT systems
    - ♦ Web-based applications
    - ♦ Increasingly also embedded applications (e.g. mobile phones)
- Why not C#,C++,Smalltalk, Ruby, Python
  - No fundamental reason
  - C# is widely used also, but tied to Microsoft
  - C++ very important, but has a lot of baggage

---

# The Java Framework

## (An Overview)

# What is Java?

---

- Programming based on objects; no global functions or variables
- Uses C++-style syntax, statements and expressions
- Developed by Sun (now Oracle)
  - Java is, first and foremost, a *specification*
    - ♦ so it is much more standardised than C++
  - Sun also developed a (reference) implementation
    - ♦ but so have others (OpenJDK, Kaffe, Jikes)
- Java programs are compiled to bytecode, which is interpreted by a virtual machine

# What is Java?

---

- Java is both language and platform
  - Language (specification)
    - ♦ Core syntax and features
    - ♦ Class declarations (API)
  - Platform (implementation)
    - ♦ Java Virtual Machine (JVM) interprets bytecodes
    - ♦ Libraries contain class implementations
    - ♦ Together = Java Runtime Environment (JRE)
- Development tools
  - **javac** compiler, **jdb** debugger, **javadoc**, etc.
  - Together with runtime = JDK/SDK (java development kit)



# Java: The Pros

---

- Portability
  - Java Virtual Machine implementations widely available (write once, run everywhere)
  - Bytecode/JIT compilation provides efficiency
  - Swing classes interface to X (Unix), Windows, Mac, etc.
- Mobility
  - Bytecodes can be downloaded (or uploaded) for execution
  - Web browsers can execute Java code - *applets*.
- Security
  - Java bytecodes can be verified and sandboxed
  - SecurityManager class provides fine control over permissions

## Java: The Cons(?)

---

- Sometimes not completely portable
  - (Write once, debug everywhere)
- Less efficient than native code
  - But JIT (Just In Time) compilation is very sophisticated now
  - If necessary, can compile to native code or make external calls to native code

## Some Features of Java

---

- No explicit pointers; only references and garbage collection
- Single inheritance; multiple inheritance through Interfaces
- Error handling using exceptions
- Standard class library (API), with abstractions for:
  - Networking
  - Graphical User Interfaces
  - Concurrency support (threads and monitors)
- Reflection (programmatic access to class structure at runtime)
- Permits remote invocation method (RMI) and Java native Interface (JNI) calls
- Automatic HTML documenting (javadoc)
- International character set support (Unicode)

# Hello World

---

```
class Hello {  
    public static void main(String[] args) {  
        System.out.println("Our first Java program");  
    }  
}
```

- Each class must live in a `.java` file with the same name
- e.g. the `Hello` class must live in `Hello.java`
- Java does not have separate header files
- Java does not have standalone functions (only methods)
  - We implement the main function using a static method

# Hello World

---

```
class Hello {  
    public static void main(String[] args) {  
        System.out.println("Our first Java program");  
    }  
}
```

- We compile the program using `javac`:  

```
% javac Hello.java
```
- This produces a *bytecode* file `Hello.class`
- This is run using `java` which executes the static main method of the specified class:

```
% java Hello  
Our first Java program
```

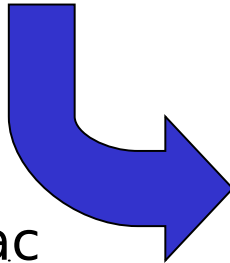
# Compiling Java Programs

---

## Hello.java

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

javac



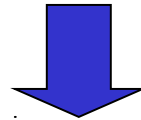
## Hello.class

```
...  
Method Hello()  
    0 aload_0  
    1 invokespecial #1 <Method java.lang.Object()>  
    4 return  
Method void main(java.lang.String[])  
    0 getstatic #2 <Field java.io.PrintStream out>  
    3 ldc #3 <String "Hello World!">  
    5 invokevirtual #4 <Method void println(String)>  
    8 return
```

# The Java Virtual Machine

---

```
class Hello {  
    public static void main() {  
        // do something  
    }  
}
```



Hello.class

Mac JVM



Hello.class

Linux JVM



Hello.class

Win JVM



## Classpath: Where's My Code?

---

- The JVM knows where to find the standard library classes
  - but what about 3<sup>rd</sup> party code (our own, or others')?
- We can tell the JVM where to look for such classes using the *classpath*
  - Can be given to `java` directly using a flag
  - Can be stored in the `CLASSPATH` environment variable

```
~/proj1% java -cp ~/lib/lib_foo ProjMain
```

```
~/proj2% setenv CLASSPATH ~/lib/lib_bar
```

```
~/proj2% java ProjMain
```



---

# Language specification

- References
- Encapsulation
- Polymorphism and Interfaces
- Inheritance and abstract classes
- Inputs/Outputs

# Objects and Classes in Java

---

- A **class** characterises common aspects of a set of objects of similar kind:
  - **Structural** characteristics (**fields**)
  - **Behavioural** characteristics (**methods**)
- Classes are used to model entities in the problem domain
- Classes should be **cohesive**: fields and methods should be defined with a logical connection
- Classes are the blueprint of objects; an object is an instance of a class
- An Object encapsulates **state** and **behaviour** and has unique **identity** (think as its position in memory)

## A Class Definition

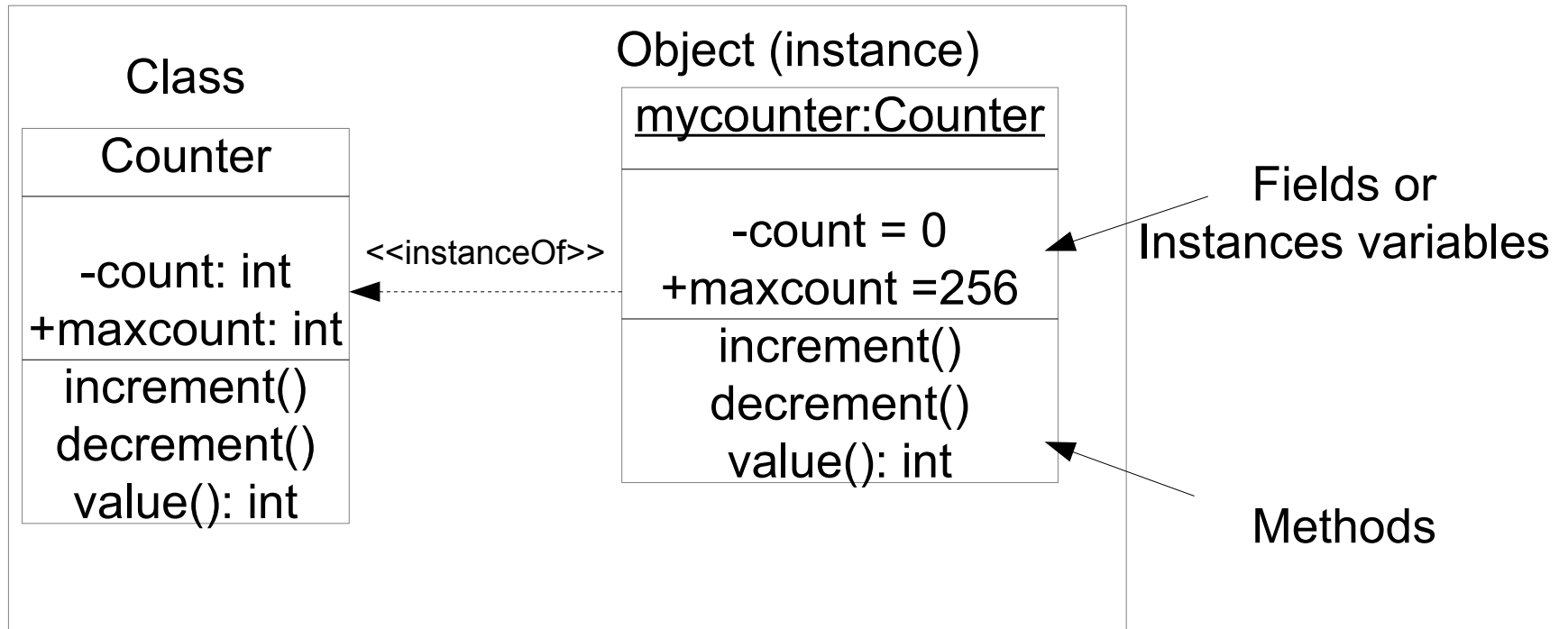
---

```
class Counter {  
    private int count = 0;  
    private static int maxcount = 256;  
    public Counter(int i) /* no initialiser list */ {  
        count = (i < 0) ? 0 :  
            ((i > maxcount) ? maxcount : i); }  
    public Counter(String s) {  
        count = Integer.parseInt(s);  
    }  
    public void increment() {  
        if (count < maxcount) count++; }  
    public void decrement() { if (count > 0) count--; }  
    public int value() { return count; }  
  
}
```

# Objects and Classes in Java

---

UML diagram



# References to Objects

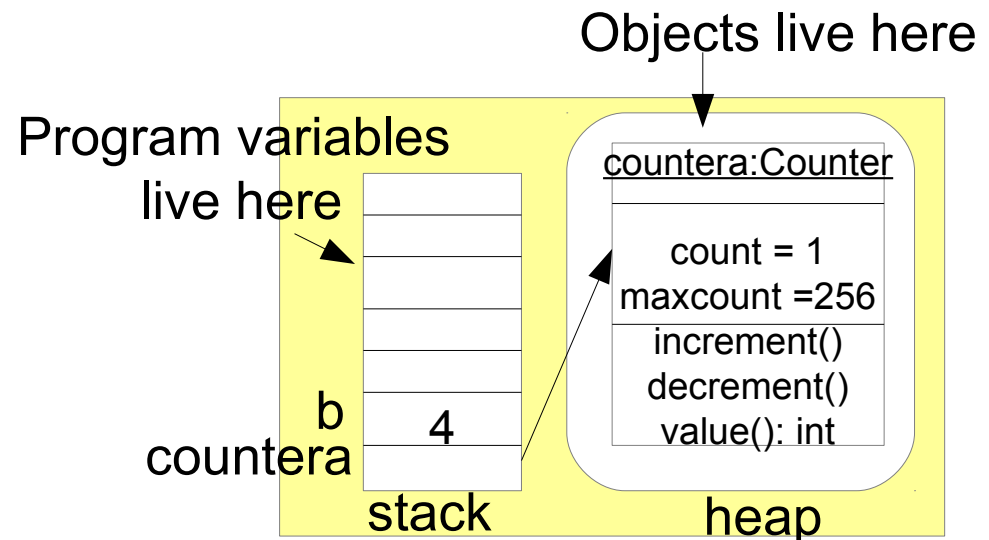
---

- A reference to an object is the position in the heap where the object is stored
- Interaction is always through a reference to the object
- `this` keyword returns a reference object to itself
- `new` keyword returns a reference which is stored in a reference variable in the declaration of the object:

```
Counter countera ; //reference variable
```

```
countera = new Counter(1) ;
```

```
int b = 4 ;
```



# Datatypes in Java

---

- In Java all variables are **typed** and stored in the program stack.
- Primitive types- Values reside in Stack
  - `boolean, char, byte, short, int, long, float, double`

1. Reserves memory in the heap for an object  
2. Creates an object using the class  
3. Runs a constructor method defined within the class  
4. Returns a reference to the object

- Reference types – References reside in Heap
  - for objects (including arrays)
    - always declared using the `new` keyword and passed by reference
  - but they can be null, and re-assigned
  - wrapper classes for primitive types: `Integer, Long, etc.`

# Declaring and Using Objects

## *Declaring objects:*

```
Counter a = new Counter(0);  
Counter b = new Counter(1);
```

## *Method invocation:*

```
a.increment();  
b.decrement();
```

## *Passing as argument:*

```
foo(a);  
a.decrement();
```

## *Assignment:*

```
a = b;           // makes a and b point at the same object
```

Careful with assignment “=” operator

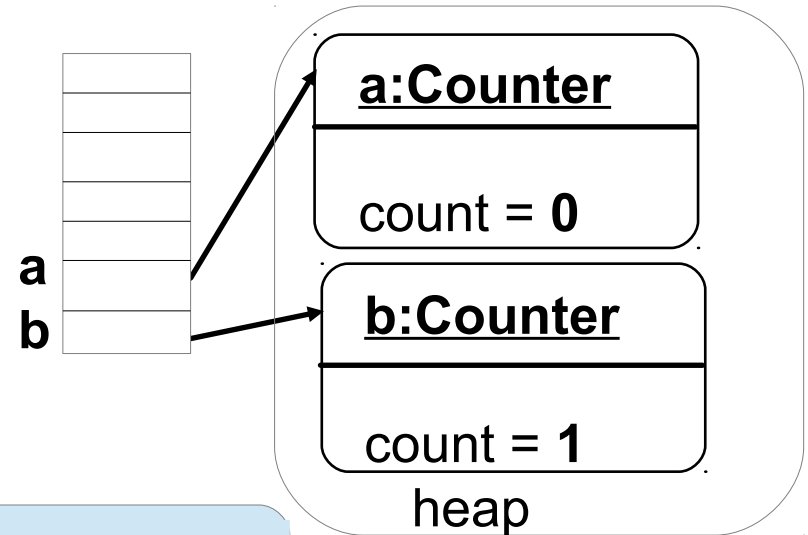
aliasing

## *Testing for equality:*

```
a == b           // tests if a and b point at the same object
```

## *Automatic garbage collection:*

```
a = null;        // may be garbage-collected if no more references left
```



All references to objects in Java are implicit pointers!

# Garbage Collection

---

- The separation of objects and references can lead to:
  - Reference variables pointing to no object (**null pointers**)
  - Object with many references to them (**aliasing**)
  - Objects with no references to them (called **inaccessible objects**)
- Java has a garbage collection process
  - Runs automatically
  - Frees up memory by removing objects that are inaccessible
  - You do not have to worry about “destruction”; leave objects in the heap, the garbage collector will remove them for you



# Operators and Control Flow

---

- Java has all (nearly all) the familiar operators from C++
  - `++`, `-`, `*=`, `? :`, `&&`, etc.
  - (obviously) no pointer derefencing operators `*`, `->`
  - no user-defined operator overloading, however
- All the usual C++ statements can be used
  - `while`, `do`, `for`, `switch`, etc.
- Enhance loop; designed for iteration through *Arrays* or *Collections*:

```
int[] numbers = {1,2,3,4,5};
for(int item : numbers){
    System.out.println("The number is: " + item);
}
```

- Labelled break and continue

```
loop: for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        if (...) { break loop; }
    }
}
```

# Arrays in Java

---

- Arrays are special kinds of object, created using

- the **new** keyword

```
int[] table = new int[128];  
Object[] objects = new Object[10];
```

- an array literal

```
int[] lookup = { 1, 2, 4, 8, 16, 32, 64, 128 };  
Object[] objs = { new A(), new B(), new C() };
```

- Array access is as expected

```
table[i] = 0;  
int i = lookup[3];
```

- But remember, no pointers!

```
// Object o = objects + 2;  
// lookup++;
```

# Arrays in Java

---

- All array objects have a **length** field:

```
for (int i = 0; i < table.length; i++)  
    table[i] = 0;
```

- The **System** object has a method for efficiently copying arrays

```
System.arraycopy(lookup, 0, table, 3, 8);
```

copies elements 0-7 from **lookup** to indexes 3-10 in **table**

- The **Arrays** object has methods for manipulating arrays
  - `Sort`, `binarySearch`, `equals`, `fill`, etc.

# Strings in Java

---

- Strings are objects, not null-terminated arrays of characters:

```
String a = "Hello";
```

```
String b = "World";
```

- String concatenation is achieved using the + operator

```
String message = a + ", " + b;
```

- The `String` class has methods for (non-destructively) manipulating strings:

- `length`, `charAt`, `indexOf`, `compareTo`, `startsWith`, `substring` and many more

- Never use `==` or `!=` to compare **String** references, always use **`equals()`**

- **StringBuffer** and **StringBuilder** are classes for efficiently dealing with *mutable* strings

- `append`, `insert`, `delete`, `replace`, `reverse`, `setCharAt`

# Packages

---

In Java, packages:

- Are collections of related classes
- Create *namespaces*
  - different classes in separate packages can have the same name (CAREFUL!)
- A package can contain other packages; can be nested to create hierarchies

To place a class in a particular package, include a **package** directive, which must be the first declaration in the file

```
package JCool.Server.Database;  
public class DataManager { ... }
```

If we want the class to be visible to code in other packages, we must declare the class `public`

# Packages

---

Java enforces file organization in directory tree, where classpath points to root:

```
H: /Java/                                ← CLASSPATH
    Jcool/
        Client/
            Gui/
                Server/
                    Database/              ← Package
                        Engine/
```

Class `JCool.Server.Database.DataManager` must live in the file `JCool/Server/Database/DataManager.java`

# Using Library Classes

---

- C++ has the **using** keyword, Java's version is **import**

```
import javax.swing.JPanel;  
...  
JPanel panel = new JPanel();  
panel.setVisible(true);  
...
```

- The `JPanel` class actually lives in the package `javax.swing`
- We can import all the classes in a package in one go

```
import javax.swing.*;
```

- Notice that this does not import classes from e.g.

```
javax.swing.text
```

- The Java compiler automatically imports all the classes from the `java.lang` and the parent package of the current class

# Comments and javadoc

---

```
/**
 * The base class for all graphics contexts
 * @author
 */
public abstract class Graphics {
    /**
     * Draws an image
     * @param img    the image to be drawn
     * @return      <i>true</i> if successful
     */
    public abstract boolean drawImage(Image img) ;
}
```





# Static Fields and Methods

---

- They are declared using the `static` keyword
- Static Fields (**class variables**) are treated as Global Variables and Static Methods as Global Methods (**class methods**)

```
class A {  
    public static int oneOnly;  
    int someMethod(){ ...}  
    static int globalMethod(){....}  
}
```

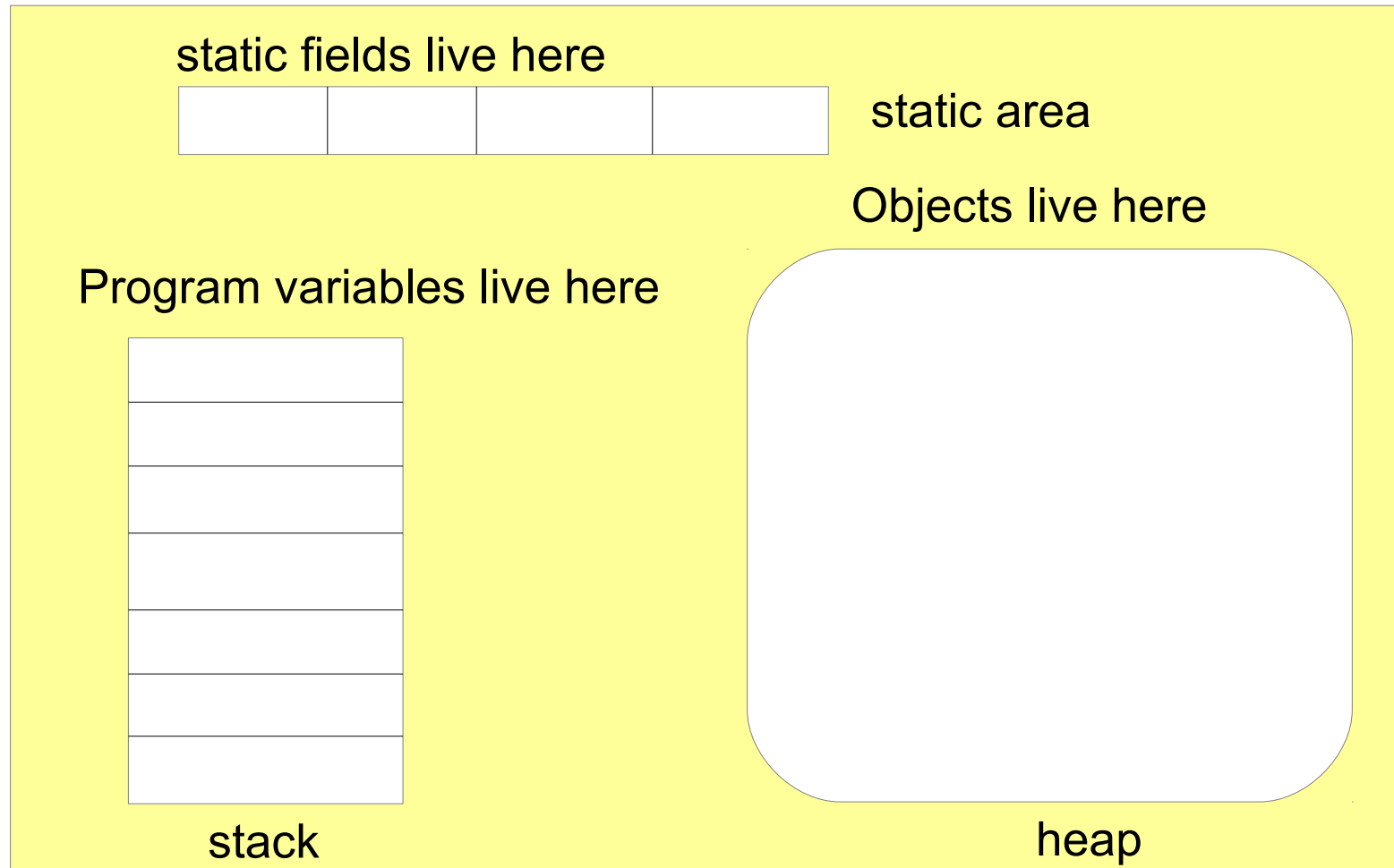
Only one static field  
is created per class  
not one per object

```
class B {  
    void foo() { System.out.println("A::foo"); }  
    public static void main(String[] args) {  
        A.oneOnly = 2;  
        A myClassA = new A();  
        int x = A.globalMethod();  
        int y = myClassA.someMethod();  
        foo(); Compile error: non-static method foo()  
        //      cannot be referenced from a static context  
    }  
}
```

You do not invoke  
`globalMethod()`  
on an instance of A

# Static Fields and Methods- view of memory

---



# Encapsulation and Visibility

---

- Classes should:
  - Give access to well-defined behaviours (methods)
  - Hide internal data representation (fields)
  - Hide algorithmic implementation details
- Classes should allow **changing internal representation without affecting users**
- **Visibility** explains who can access methods, fields and classes. It is a mechanism for implementing **encapsulation**.
- Methods and fields can be declared:
  - **public**: anyone can access
  - **package**: (the default) anyone from within the package
  - **private**: accessible only within the class
  - **protected**: related to inheritance



encapsulation

## Immutable objects (value objects)

---

Declaring fields `final` the values cannot changes after the construction of this object

```
public class Rectangle {  
    private final int width, height;  
  
    public Rectangle(int width, int height) {  
        this.width = width; // one and only  
        this.height = height; // assignment to fields  
    }  
  
    public int getWidth() { return width; }  
  
    public Rectangle rectangleBySettingWidth(int width) {  
        return new Rectangle(width, getHeight());  
    } // similar for height  
}
```

## Interfaces- Case study

---

- Imagining we design a document management application that manages at start various kind of page elements:
  - Tex box- has width, height, unique label and maximum number of characters
  - Menu - has width,height, unique label and list of options
- How can we find the height of the tallest page element?
- Let's add another page element...
  - Image – has width, height, unique label and filename associate to

## Interfaces- Case study

---

- Problems with this design:
  - Duplicate code everywhere
  - `DocumentManager` has to be explicitly aware of all the different sorts of page elements that exist
  - If we introduce a new page element, we need to change `DocumentManager`
  - Makes third parties difficult to contribute
  - If another actions such as determine one page element is taller than another, one approach we would have to overload 'tallerThan' for each pair of types  $N$  kinds of page element  $\rightarrow N^2$  'tallerThan' methods

# Interfaces

---

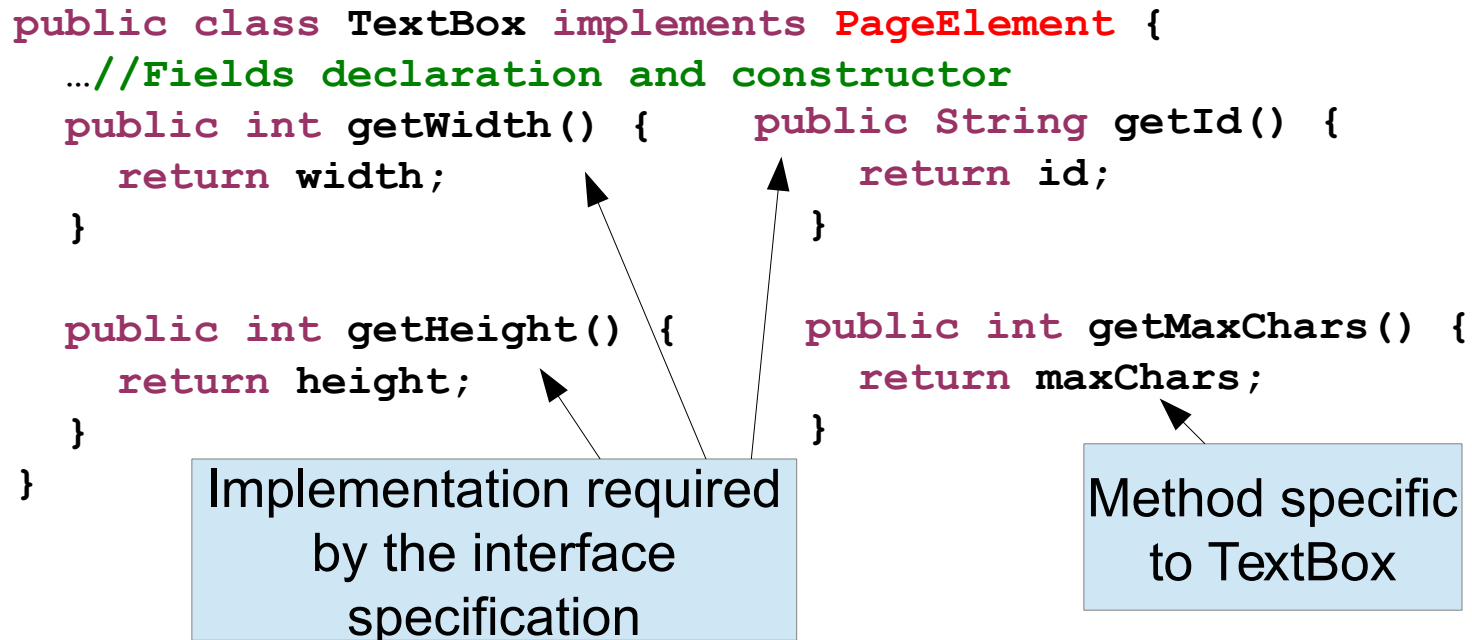
```
public interface PageElement {  
    public int getWidth();  
    public int getHeight();  
    public String getId();  
}
```

- Interfaces define a group of methods to be implemented
- Methods are declared without bodies
- Interfaces, like classes, induce named types
- Interfaces does not contain fields, however
  - Interfaces may contain constant fields
  - They are implicitly `public static final` when the interface is implemented
- Interfaces can extend each other



# Implementing Interfaces

---

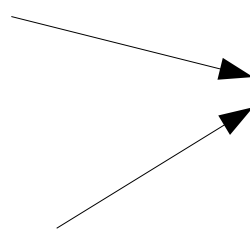


- Methods in a class that implement interface must be public
- Interfaces **decouple**(separate) the specification from the implementation
- A class can use any class that implements the interface through the interface type (**polymorphism**)

# Multiple Interfaces

---

```
public interface AlarmClock{
...
}
public interface Clock{
...
}
```



The diagram consists of two arrows. The first arrow originates from the `AlarmClock` interface definition and points to the `AlarmClock` in the `SmartPhone` class's `implements` clause. The second arrow originates from the `Clock` interface definition and points to the `Clock` in the same `implements` clause. This visually demonstrates that the `SmartPhone` class implements both interfaces.

```
public class SmartPhone
implements Clock, AlarmClock{
...//Implements at least
specified methods in both
interfaces
}
```

- Classes are allowed to implement multiple interfaces
- Multiple methods with the same name (**Name conflict**):
  - Different signatures:overloading
  - Same signatures and same return type: same method!
  - Same signature and different return type: //compile error

# Apparent and actual types

---

- The **apparent type** of an object is what the compiler statically can infer
- The **actual type** of an object is what the object really is
- RULES:
  - Interfaces or classes can be apparent type
  - Only classes can be actual type
  - Methods and fields available according to apparent type
  - In inheritance **apparent type** must be a super type of **actual type**
- Java checks these rules at compile time
  - Java is a **strongly typed** language
  - It does static type checking

## Using interfaces: late binding and polymorphism

---

- If a reference variable has interface type, we can use the same way as a class, however:
  - Only call to methods that are specified by the interface
  - Cannot access fields through an interface: the classes that implements the interface
- When we call a method on an interface, the actual method invoke is decided at runtime through **late binding**

# Inheritance in Java

---

- Java has **single** inheritance only (for multiple inheritance use interfaces as classes are allowed to implement multiple interfaces)
- Inheritance is indicated by the `extends` keyword

```
class Foo extends Bar { ... }
```

- Derived objects:
  - **extend** their superclass
  - **Inherited** all members of superclass (including private)
  - Can **override** methods of superclass
  - Can **access** public/protected fields of superclass
  - Can **call** public/protected methods of superclass
- For example in Java:
  - Classes inherit from `Object` by default

```
class Counter /* extends Object */ { ... }
```
  - The `Object` class contains a set of 'standard' methods
    - e.g. `clone()`, `equals(Object obj)`, `toString()`

# Super

---

```
public Circle(int radius) {  
    super();  
    this.radius = radius;  
}
```

- Used to call superclass' constructor
- Must be called the first line
- Can have different signatures (super(), super(arg1, arg2),)
- super() is called implicitly if no other constructor is called
- Used to call an overridden method for a superclass

```
super.superclassmethod();
```

## Visibility

---

|                    | public     | protected  | package<br>(default)                     | private    |
|--------------------|------------|------------|--|------------|
| Within<br>class    | <b>Yes</b> | <b>Yes</b> | <b>Yes</b>                               | <b>Yes</b> |
| Within<br>package  | <b>Yes</b> | <b>Yes</b> | <b>Yes</b>                               | <b>No</b>  |
| Within<br>subclass | <b>Yes</b> | <b>Yes</b> | <b>No</b><br>(unless in same<br>package) | <b>No</b>  |
| Anywhere           | <b>Yes</b> | <b>No</b>  | <b>No</b>                                | <b>No</b>  |

# Abstract Classes

---

- *Pure* (virtual) methods are declared using the `abstract` keyword
- Unlike classes, they cannot be instantiated
- No object can have as **actual type** the type of abstract class
- Why to use them?
  - We want to prevent users from handling objects that are too generic (**BasicPageElement**)
  - We cannot give full implementation for the class (**Shape**)



# Rules - Abstract Classes

---

```
public abstract class A {  
    public abstract void foo();  
}
```

```
public class B extends A {  
    //Compile-error. Class B must  
    //implement abstract method foo  
}
```

```
public abstract class B extends A {  
    //ok. Subclasses of B must implement  
    //foo (or declared abstract)  
}
```

```
public abstract class B extends A {  
    //ok. Subclasses of B must implement  
    //foo and bar(or declared abstract)  
    abstract void bar();}
```

## Rules - Abstract Classes

---

```
public class A {  
  
}  
public abstract class B extends A {  
    //ok, but uncommon  
}  
public abstract final class A {  
    //Compile-error class cannot  
    // be abstract and final  
}
```

---

# Input/Output

# Input/Output in Java

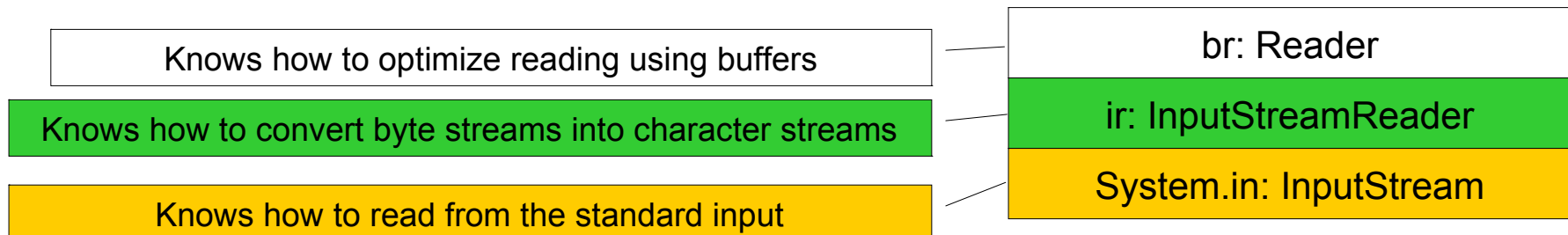
---

- Interacting with the user
  - `System.out` is a global `PrintStream` object: outputs to the screen  
**`System.out.println("Hello");`**
  - `System.in` is a global `InputStream` object: reads from the keyboard
- The `java.io` package or `java.util.Scanner` contains library classes for performing input and output actions
- How to read a line from keyboard?
  - **`BufferedReader`** class can read characters and lines
  - How can we build a **`BufferedReader`**? We need an **`InputStreamReader`**
  - How can we build an **`InputStreamReader`**? We need an **`InputStream`**
  - How can we build an **`InputStream`**? We have one **`System.in`**

# Input/Output in Java

---

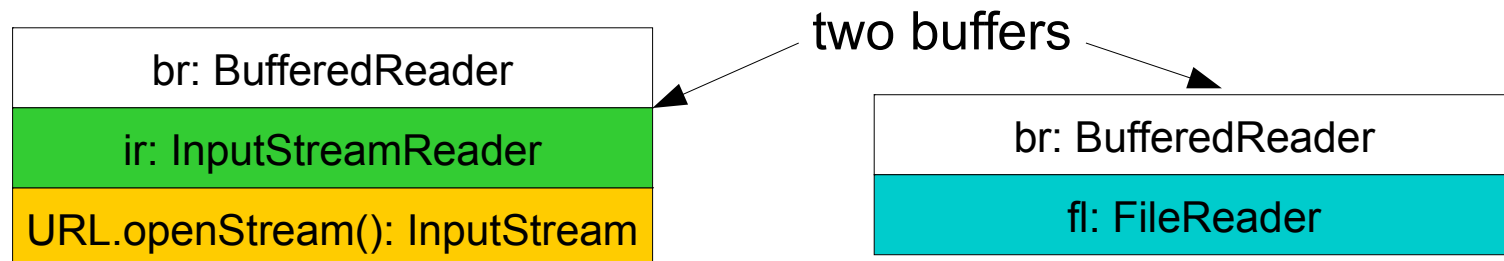
- The Java I/O classes are designed to build on top of one another to provide:
  - Separation of concerns
  - Efficiency
  - Flexibility- we can use different input streams to read from:
    - The web
    - A file
    - Other complex devices



# Input/Output in Java

---

- We can have multiple objects of a same class with different state
  - Several files being read simultaneously
  - Several URL being read simultaneously
- Objects encapsulate their state and behaviour: we can have multiple objects performing radically different tasks
  - Several BufferedReader objects reading from files, URLs and the standard input simultaneously
- Example: a program that checks if a web page is up to date by comparing it with its local copy of the html file



# Input/Output in Java

- Input/OutputStream classes:

- System.in
- System.out
- DataInputStream: reads primitive datatypes from its source
- DataOutputStream: writes primitive datatypes in a portable way
- ObjectInputStream: reads “serializable” objects from an InputStream
- ObjectOutputStream: writes “serializable” objects to an OutputStream

- InputStreamReader/OutputStreamWriter classes:

- FileReader: file as source
- FileWriter: file as a sink

- Reader/Writer classes:

- CharArrayReader: byte[] as source
- CharArrayWriter: accumulates writes and provides data as a byte array
- BufferedReader: will read enough data to fill an internal buffer
- BufferedWriter
- StringWriter
- PrintWriter

| Reader/Writer                        |
|--------------------------------------|
| InputStreamReader/OutputStreamWriter |
| InputStream/OutputStream             |

---

# Generics and Collections

(<http://docs.oracle.com/javase/tutorial/extra/generics>)

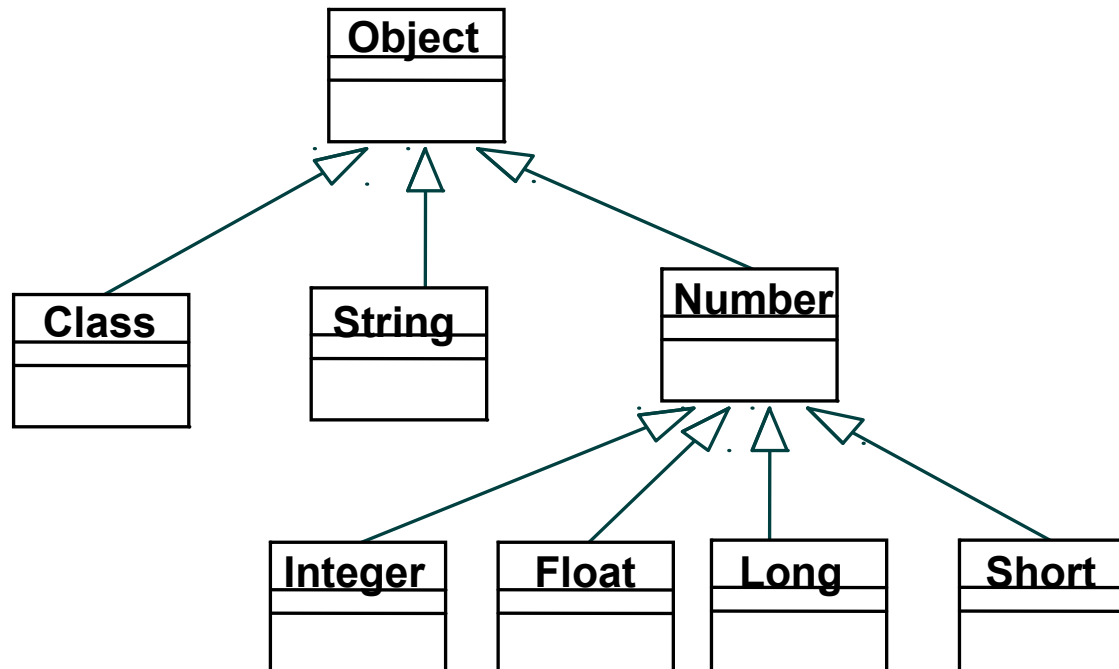
- Casting and Object equality
- Generics methods
- Generics and subtyping
- Wildcards
- Inheritance and generics
- Collections



# Casting

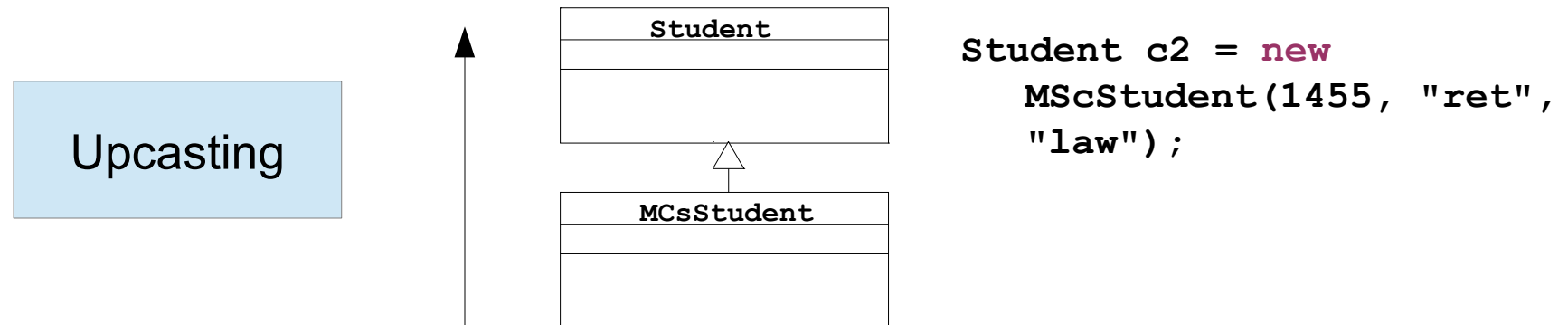
---

- Looking at Java Development Kit Documentation:
  - Every Java class is a subclass of Object
  - Object is the superclass
  - Being an Object subclass is not declared explicitly

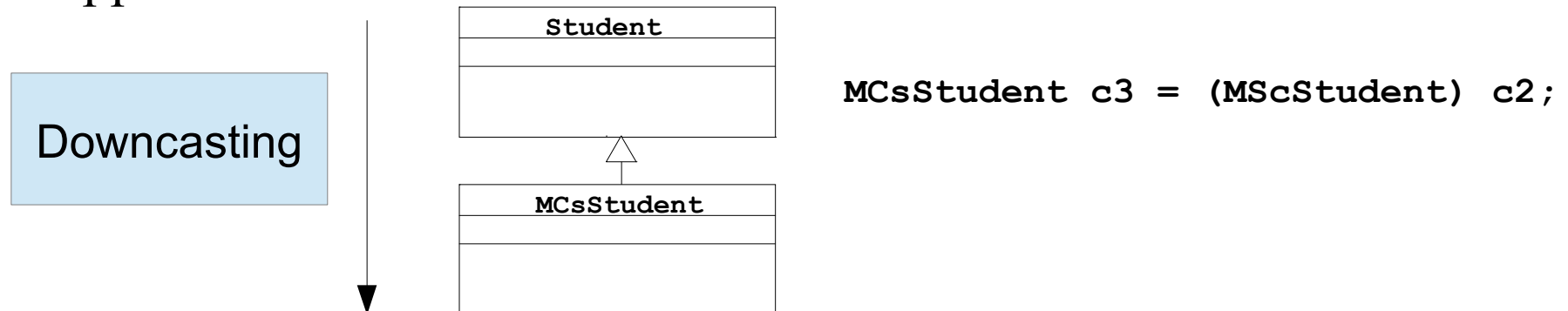


# Casting

- **Upcasting** or widening type: the process of treating an object of type subclass as if it was of type superclass.
  - **Upcasting** is done automatically, it cannot fail.



- **Downcasting** or narrowing type: to reduce the type of the apparent class to be able to call certain methods.



# Casting

---

- Downcasting is an explicit type conversion
- Some downcasts cannot be verified at compile time, may produce run-time errors: `ClassCastException`
- To avoid `ClassCastException` use `instanceof` to discover the actual type of an object (it takes care of null `instanceof T`)

```
if (c1 instanceof MScStudent) {  
    MScStudent c3 = (MScStudent) c1;  
}
```

- However, sometimes to use `instanceof` may indicate poor design, better to use subclassing

# Object equality

---

- As discussed class Object provides equals method:

```
public boolean equals(Object obj);
```

- The equals method must be:
  - Reflexive: for any object, o.equals(o) must be true
  - Symmetric: for any two objects, o1.equals(o2) holds if and only if o2.equals(o1)
  - Transitive: for any three objects, if o1.equals(o2) and o2.equals(o3), it must be the case that o1.equals(o3)
  - Consistent: if we keep asking whether o1.equals(o2) and neither o1 nor o2 changes, the result should stay the same
  - o.equals(null) must be false
- o1.equals(o2) if and only if o1.hashCode() == o2.hashCode()
- Therefore, if you override equals() you must override hashCode()

## Object equality- Override annotation

---

- Writing a basic equals method:
  - Check incoming object has appropriate type
  - Downcast incoming object
  - Compare fields
- Good practice – specify the intention of override a superclass method by typing “@Override” on top of the method to override
- If you define a method with `final` keyword the method cannot be overridden
- If you define a class with `final` keyword the class cannot be subclassed (**immutable** classes)
- Private methods cannot be overridden

# Generics

---

- It is the Java version of C++ Templates with some differences.  
In Java:
  - Only type parameters (no constant or function parameters)
  - Generics allow for **subtyping**
- Constructor declarations do not use parameter specification
  - `public Pair<S, T>(S first, T second){...}` // **WRONG**
  - `public Pair(S first, T second){...}` // **CORRECT**
- Constructor calls reference variable declarations **do** use parameter specification
  - `Pair p;` // **WRONG**
  - `Pair<String, Integer> p;` // **CORRECT**
  - `new Pair();` // **WRONG**
  - `new Pair<String, Integer>();` // **CORRECT**

# Generics

---

- Actual parameter of a generic must be a **reference type**
  - `Pair<int, int> //WRONG`
  - `Pair<Integer, Integer> //CORRECT`
- Illegal statement with formal type parameters:
  - `new T(); //WRONG: generic parameter cannot denote a constructor`
  - `new T[2]; //WRONG: generic parameter cannot denote an array constructor`
- Arrays of generic classes are not allowed
  - `Pair<String, String> values = new Pair<String, String> [2]; //WRONG`
  - `Pair<String, String> values = new Pair<String, String> { ("one", "two"), ("three", "four") }; //WRONG`
- However, you can write the following, although you will get a warning while compiling:
  - `Pair<String, String>[] values = new Pair[] { new Pair<String, String>("one", "two"), new Pair<String, String>("3", "4") }`

# Generic Classes

---

```
public class Pair<S, T> {  
  
    private S first;  
    private T second;  
  
    public Pair(S first, T second) {  
        this.first = first;  
        this.second = second;  
    }  
    public S getFirst() {  
        return first;  
    }  
    public void setFirst(S first) {  
        this.first = first;  
    }  
    public T getSecond() {  
        return second;  
    }  
    public void setSecond(T second) {  
        this.second = second;  
    }  
}
```



## Generic methods

---

- Add `<S,T,...>` before **return type** (int, void,...) of the method to indicate that the method should work for arbitrary types S,T,etc
- Then use freely the types S,T inside the method:

```
public static <S,T> Pair<S,T> makePair(S first, T second){  
    return new Pair<S,T>(first,second);  
}  
public static <S> Pair<S,S> makeHomogeneousPair(S first, S  
    second){  
    return new Pair<S,S>(first,second);  
}  
public static <S,T> T lookPair(S key, Pair<S,T>[] elems, T  
    generic){  
    for( Pair<S,T> elem : elems){  
        if(key.equals(elem.getFirst())){  
            return elem.getSecond();  
        }  
    }  
    return generic;  
}
```

return type

## Generics and subtyping

---

- If String extends Object, is Set<String> a subtype of Set<Object>? Can we swap safely Set<String> for Set<Object>?

```
static double maxArea(Set<Shape> shapes) { ... }
```

```
public static void main(String[] args) {
```

```
Set<Circle> circles = new HashSet<Circle>();
```

```
...
```

```
double d = maxArea(circles); // Seems OK, but gives  
compile error
```

```
Shape rectangle = new Rectangle(...);
```

```
addOne(circles, rectangle); // Certainly not OK
```

```
}
```

## Generics and subtyping

---

- Our intuition say Yes, but in general the answer is NO!
- The general rule is:

If C is a generic class, and B extends A,  
then C<B> is **NOT** a subtype of C<A>

- We can also say that C<A> and C<B> are not **covariant**

# Wildcards

---

- Suppose we wish to print an arbitrary set of objects

```
public class WithoutWildcards {
```

```
    public static void printSet(Set<Object> s) {  
        for (Object o : s) {  
            System.out.println(o);  
        }  
    }
```

Too specific – only accepts values  
with exactly type `Set<Object>`

```
    public static void main(String[] args) {  
        Set<Shape> shapes = new HashSet<Shape>();  
        ...  
        printSet(shapes);  
    }  
}
```

Does not compile – `Set<Shape>` is  
**not** a subtype of `Set<Object>`

## Wildcards

---

Set<?> vs Set<Object>

- We can add a `String` to a `Set<Object>` because `String` is a subtype of `Object`
- We *cannot* add a `String` to a `Set<?>` because `?` could denote *anything* (e.g. we might really have a `Set<Integer>`)
  - `Set<?>` is **not** a subtype of `Set<T>` for any `T`
- Is there anything that we can add to a `Set<?>`
- Is there any type we can guarantee for elements retrieved from a `Set<?>`

# Wildcards

---

- Suppose we want to find the maximum area over a set of shapes

Too specific – only accepts values with exactly type `Set<Object>`

```
public class WithoutBoundedWildcards {  
    public static void printColourShape(Set<Shape> shapes) {  
        for (Shape s : shapes) {  
            System.out.println(s.getColour());  
        }  
    }  
    public static void main(String[] args) {  
        Set<Circle> circles = new HashSet<Circle>();  
        ...  
        printColourShape(circles);  
    }  
}
```

Does not compile – `Set<Circle>` is **not** a subtype of `Set<Shape>`

# Bounded wildcard

---

- Does a wildcard help?

Does not compile – we cannot retrieve Shape objects from a Set<?>

```
public class WithoutBoundedWildcards {  
    public static void printColourShape(Set<?> shapes) {  
        for (Shape s : shapes) {  
            System.out.println(s.getColour());  
        }  
    }  
    public static void main(String[] args) {  
        Set<Circle> circles = new HashSet<Circle>();  
        ...  
        printColourShape(circles);  
    }  
}
```

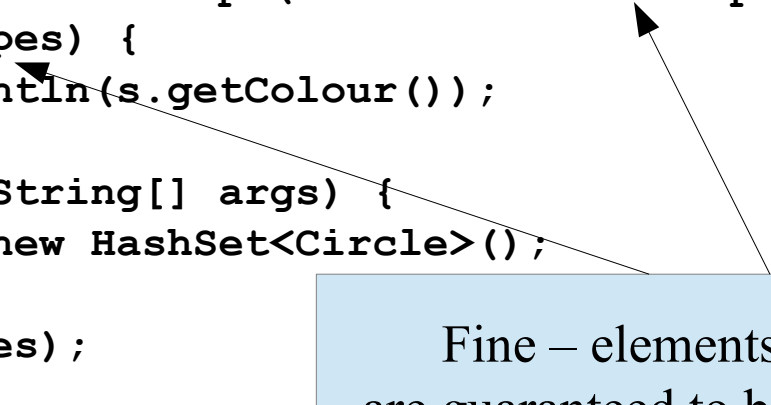
However, OK – Set<Circle> is a subtype of Set<?>

# Bounded wildcard

---

- We use a bound on the type parameter
  - `Set<? extends Shape>` is a set whose elements are some (unknown) subclass of `Shape`

```
public class WithBoundedWildcards {  
    public static void printColourShape(Set<? extends Shape> shapes) {  
        for (Shape s : shapes) {  
            System.out.println(s.getColour());  
        }  
    }  
    public static void main(String[] args) {  
        Set<Circle> circles = new HashSet<Circle>();  
        ...  
        printColourShape(circles);  
    }  
}
```



Fine – elements of shapes  
are guaranteed to be `Shape` objects

OK – `Set<Circle>` is a  
subtype of `Set<? extends Shape>`



## Wildcards

---

Set<? extends Shape> VS Set<Shape>

- We can add a Circle to a Set<Shape> because Circle is a subtype of Shape
- We *cannot* add a Circle to a Set<? extends Shape> because ? could denote *anything* (e.g. we might really have a Set<Rectangle>)
- Is there anything that we can add to a Set<? extends Shape>?
- Is there any type we can guarantee for elements retrieved from a Set<? extends Shape>?

# Bounds for Generic Type Parameters

---

- We make the method generic, placing a (named) bound on the return type
  - The method can only be instantiated by types `T` that extend `Shape`

```
public class BoundedGenerics {  
    public static <T extends Shape> T largest(Set<T> shapes) {  
        T result = null; boolean first = true;  
        for (T s : shapes) {  
            if (first) { result = s; first = false; }  
            else if (s.area() > result.area()) result = s;  
        }  
        return result;  
    }  
    public static void main(String[] args) {  
        Set<Circle> circles = new HashSet<Circle>();  
        ...  
        Circle largestCircle = largest(circles);  
    }  
}
```

OK – Circle extends Shape

OK – return type of largest is T,  
which in this invocation is  
Circle

# Bounds for Generic Type Parameters

---

- Suppose we want to *return the element* with the largest area

```
public class BoundedWildcardsNotGoodEnough {  
    public static Shape largest(Set<? extends Shape> shapes) {  
        Shape result = null; boolean first = true;  
        for (Shape s : shapes) {  
            if (first) { result = s; first = false; }  
            else if (s.area() > result.area()) result = s;  
        }  
        return result;  
    }  
    public static void main(String[] args) {  
        Set<Circle> circles = new HashSet<Circle>();  
        ...  
        Circle largestCircle = largest(circles);  
    }  
}
```

Does not compile – largest  
returns a Shape, **not** a Circle

# Bounds for Generic Type Parameters

---

- We make the method generic, placing a (named) bound on the return type
  - The method can only be instantiated by types `T` that extend `Shape`

```
public class BoundedGenerics {  
    public static <T extends Shape> T largest(Set<T> shapes) {  
        T result = null; boolean first = true;  
        for (T s : shapes) {  
            if (first) { result = s; first = false; }  
            else if (s.area() > result.area()) result = s;  
        }  
        return result;  
    }  
    public static void main(String[] args) {  
        Set<Circle> circles = new HashSet<Circle>();  
        ...  
        Circle largestCircle = largest(circles);  
    }  
}
```

OK – Circle extends Shape

OK – return type of largest is T,  
which in this invocation is  
Circle

# Inheritance and Generics

---

- Common for a class to extend a generic class or implement a generic interface, reminding generic (Java documentation)

```
public abstract class AbstractSet<E> implements Set<T>{  
    //Provides abstract implementation of various 'set' algorithms  
}
```

```
public class HashSet<E> extends AbstractSet<E>{  
    //Implements Set using a hashtable  
}
```

```
public class LinkedHashSet<E> extends HashSet<E> {  
    //Unlike plain HashSet<E>, provides guaranteed  
    //iteration order  
}
```

# Inheritance and Generics

---

- Also useful for extending/implementing class to be specific

```
public interface Comparable<T>{  
    int compareTo(T Obj);  
}
```

```
public class UniDegrees implements Comparable<UniDegrees>{  
    public int compareTo(T Obj){...};  
}
```

```
public static void main(String[] args) {  
    UniDegrees<UniDegrees> dg; //WRONG  
    UniDegrees dg; // CORRECT- this class is not generic  
}
```

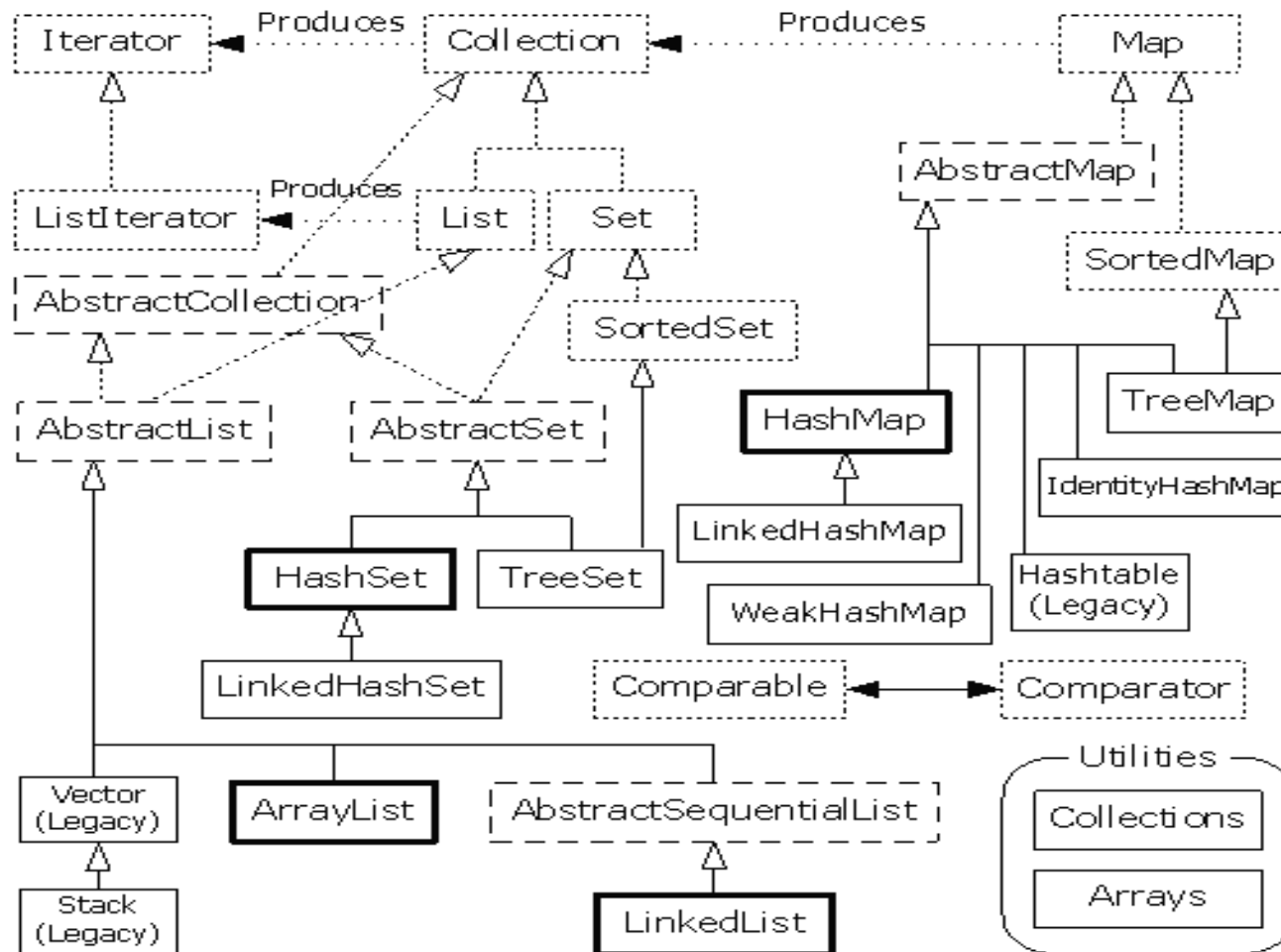
# The Collections Framework

---

- Java has a large library of container classes and interfaces
- A container allows objects to be stored and retrieved
- All elements of the framework are *generic*
- There are different types of collections according to the additional services and behaviour implemented

|       | Ordered | Duplicates  |
|-------|---------|---|
| Sets  | NO      | NO  |
| Lists | YES     | YES   |
| Maps  | NO      | <i>(key, value) pairs</i><br><i>No duplicate keys</i> |

# Container class hierarchy





# Java.util.Collection (Interface)

---

```
public interface Collection<E> {  
  
    boolean add(E o);  
    boolean remove(Object o); // Why object and not E?  
    boolean contains(Object o);  
    boolean addAll(Collection<? extends E> c)  
    boolean isEmpty();  
    Iterator<E> iterator();  
    int size();  
  
    // ... many other methods  
}
```

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();  
}
```

- All collection classes also implement Iterable<E>
  - Defines an iterator() method which returns an Iterator<E>

# Iterating Through Collections

---

```
List<String> stringList = new ArrayList()
...
Iterator<String> it = stringList.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

- However, you could also use the Enhance For loop (for-each syntax) with any class that implement Iterable

```
List<String> stringList = new ArrayList()
...
for (String s : stringList) {
    System.out.println(s);
}
```

## More Specialised Kinds of Collection

---

- `List<E>` allows fine control over the position of elements
  - Provides search operations
  - May contain duplicate elements
- `Set<E>` does not allow duplicate elements
  - uses the `equals(Object o)` method to determine equality
  - Allows a null element
- `Map<K, V>` stores pairs<*key,value*>
  - Not a Collection subclass
  - Keys are unique
  - `V put(K key, V value)`
  - `V get(Object key)`
  - `Set<K> keySet()`
  - `Collection<V> values()`

## More Specialised Kinds of Collection

---

- `SortedSet<E>` and `SortedMap<K, V>` maintain (total) orderings on their elements
- `Queue<E>` and `Deque<E>` provide methods for processing elements in FIFO (first in first out) manner
  - `E peek()`
  - `E poll()`

## Some Concrete Collection Classes

---

- Java provides some default implementations for each of the interfaces in the Collections Framework, e.g.
  - `ArrayList<E>` implements `List<E>` as a resizable array
  - `LinkedList<E>` implements `List<E>` as a (doubly) linked list
  - `HashSet<E>` and `HashMap<K, V>` implement `Set<E>` and `Map<K, V>` respectively using hash tables
  - `TreeSet<E>` and `TreeMap<K, V>` implement `Set<E>` and `Map<K, V>` respectively using red-black trees
  - `LinkedHashSet<E>` and `LinkedHashMap<K, V>` implement `Set<E>` and `Map<K, V>` respectively using hash tables, and also maintain a linked list so the elements can be iterated through in insertion order

# Sorted Collections

---

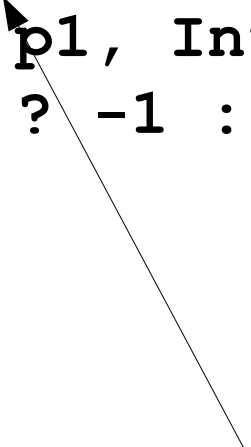
- In C++, sorted collections maintain an order on their elements
  - By using an overloaded version of the < (less than) operator
  - By taking a comparison function as a template parameter
- In Java, we cannot overload operators, and there are no function parameters, so
  - Elements of the sorted container must implement the `Comparable<E>` interface
    - `int compareTo(E e)`
  - We can provide an object to the constructor which implements the `Comparator<E>` interface
    - `int compare(E e1, E e2)`

## Sorted Collections

---

- Alternatively, we can pass the constructor a **Comparator** object of an appropriate type

```
SortedSet<SetIntPair> s = new  
TreeSet<IntPair>(new Comparator<IntPair>() {  
    public int compare(IntPair p1, IntPair p2) {  
        return p1.fst() < p2.fst() ? -1 :  
            (p1.fst() > p2.fst() ? 1 :  
             p1.snd() - p2.snd());  
    }  
}) ;
```



*Anonymous* class: it is an expression (does not have a name),  
i.e. you define the class in another expression.

In this case, it implements the `Comparator<IntPair>` interface.  
It is necessarily an *inner* class, which is a class declared inside another class

---

# Exceptions

(Dealing with Abnormal Execution)

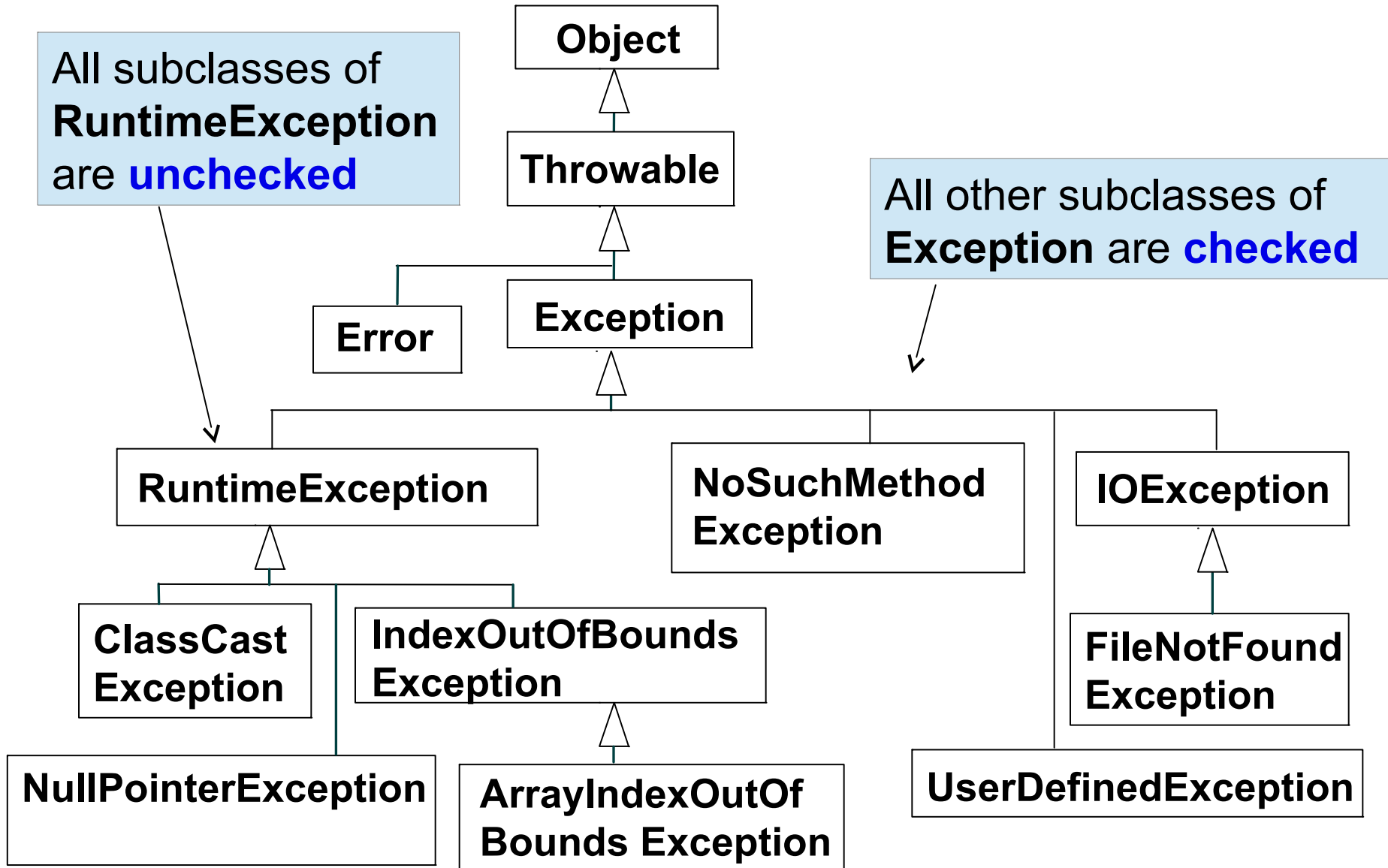


# Exceptions

---

- Exceptions and exception handling support the development of *robust* programs
  - Report and detect unexpected, erroneous situations
    - File not found / inaccessible
    - Run out of memory
    - Network unavailable
  - We can recover from the error, or exit the program safely
- Exceptions are objects; they have type and are instantiated using **new**.
- Exceptions can be defined with custom structure, operations, constructors by extending any predefined exception type.
- **Exception** class extends the **Throwable** class
  - **Exception** is the parent class of all exceptions
  - **Throwable** is also extended by **Error** and its subclasses

# Java exception class hierarchy



# Checked and unchecked exceptions and errors

---

- Checked exceptions are declared explicitly in **throws** part of method signature.
  - The compiler verifies that such exceptions are either caught or thrown by all calling methods
- Unchecked exceptions do not need to be declared in signature method that throws them
  - You can catch them, but it is not compulsory
  - If not caught, these exceptions are automatically propagated up to call stack
  - If never caught: program terminates
- The Error throwable class represent serious errors
  - Exceptions that application should not try to catch such as error in the JVM
    - StackOverflowError
    - OutOfMemoryError
  - Rarely used. Mostly to terminate programs with customised failure modes

## Exceptions methods

---

- `Throwable` class specifies basic operations for all exceptions and errors:
  - `String getMessage()` : Exceptions may contain human-readable information
  - `Throwable getCause()` : Exceptions may be caused by other exceptions and chained together
  - `void printStackTrace()` : We can see information about the chain of method calls that led to this exception being thrown. This method prints standard error output the stack of the method calls at point of creation

# Exceptions and throwing Exceptions: Java syntax

---

- For the provider method

```
<return-type> methodName(<arguments>) throws <exception list>{...}  
  
<exception list> = exceptiontype1,...,exceptiontypeN
```

- When calling a method (catching exceptions)

```
try{  
    //call the method  
}catch(ExceptionType1 name){  
    ...  
}catch(ExceptionTypeN name){  
    ...  
}finally{  
    //code here always gets executed regardless  
}
```

- Throwing exceptions is easy. Use the **throw** keyword follow the **new** object Exception instance.
- You can **throw** anything that extends **Throwable**

## Writing Exception classes

---

- If we want to create an Exception class
  - This new exception overrides `getMessage()` method

```
public class UnknownStudentException extends Exception {  
    @Override  
    public String getMessage() {  
        return "Invalid Student login (" +  
            super.getMessage() + ")";  
    }  
}
```

## Try/catch and inheritance

---

- `catch (Exception e) {...}` catches any exception which derives from (and including) `SomeException`

```
try {  
    ...  
}  
catch (Exception e) {  
    //any (non-Error) exception will be caught here  
} catch (IOException e) {  
    //unreachable, because an IOException is an Exception!  
}
```

# Method overriding: narrowing exception types

---

- This is allowed in Java:

```
public class AException extends Exception{...}
```

```
public class BException extends AException{...}
```

```
public class C{  
    public void foo() throws AException {...}  
}
```

```
public class D extends C{  
    @Override  
    public void foo() throws BException {...}  
}
```

- In other words, Java supports covariance with respect to **throws**



---

# Graphical User Interfaces, Threads and Networking

- Java Swing library
- Threads
- Sockets

# Swing: The Java GUI Framework

---

- Java has an extensive library, called Swing, for creating GUI applications
  - Oracle has recently developed JavaFX, but this builds on top of Swing and so it will be useful to learn Swing first
- It is very quick to build applications with:
  - Windows
  - Buttons
  - Menus
  - Customisable layouts
  - And many more features ...
- We will look at how to program some basic features in Swing
  - And on the way introduce Java's functionality for programming *concurrency*

# JFrame: Creating Windows

---

- The basic class for implementing windows is `JFrame`
  - We can create instances of `JFrame` and set its properties
  - Or we can create a subclass of it if we want more specialised behaviour

```
JFrame myFrame = new JFrame("This is my window");
```

This goes in the title bar

```
myFrame.setSize(1024, 768);
```

What size the window should be

```
myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

What should happen when the window is closed – we can exit the application, hide the window, or even do nothing!

```
myFrame.setVisible(true);
```

Make the window visible

# A Basic Application

---

- An application can be implemented as any ordinary class with a main method that creates a window

```
public class MyApp implements Runnable {  
    public MyApp() {}  
    public void run() {  
        JFrame frame = new JFrame("A Basic GUI");  
        myFrame.setSize(1024, 768);  
        myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        myFrame.setVisible(true);  
    }  
    public static void main(String[] args) {  
        javax.swing.SwingUtilities.invokeLater(new MyApp());  
    }  
}
```

# A Basic Application

---

- An application can be implemented as any ordinary class with a main method that creates a window

```
public class MyApp implements Runnable
{
    public MyApp() {}
    public void run() {
        JFrame frame = new JFrame("A Basic GUI");
        myFrame.setSize(1024, 768);
        myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        myFrame.setVisible(true);
    }
    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new MyApp());
    }
}
```

This method will create the window for the application and display it

Because the GUI needs to run independently alongside the main application, Swing runs it using its own *Thread*, which is created using the `Runnable` object passed as the argument to the `invokeLater` method.

# Components

---

- Windows contain various different *components*
- The basic superclass of components in Swing is `JComponent`
- Many useful GUI classes are subclasses of `JComponent`
  - `JLabel`
  - `JButton`
  - `JMenu`, `JMenuBar`, `JMenuItem`
  - `JPanel`
  - `JFileChooser`
- Some components may contain others (e.g. `JPanel`, `JMenu`)
- Other components are basic entities (e.g. `JLabel`)

# Displaying A Label

---

```
JFrame myFrame = new JFrame("This is my window");  
myFrame.setSize(1024, 768);  
myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
JLabel myLabel  
    = new JLabel("My Label", SwingConstants.CENTER);
```

The text of the label

```
mMyFrame.add(myLabel);
```

The horizontal  
alignment of the text

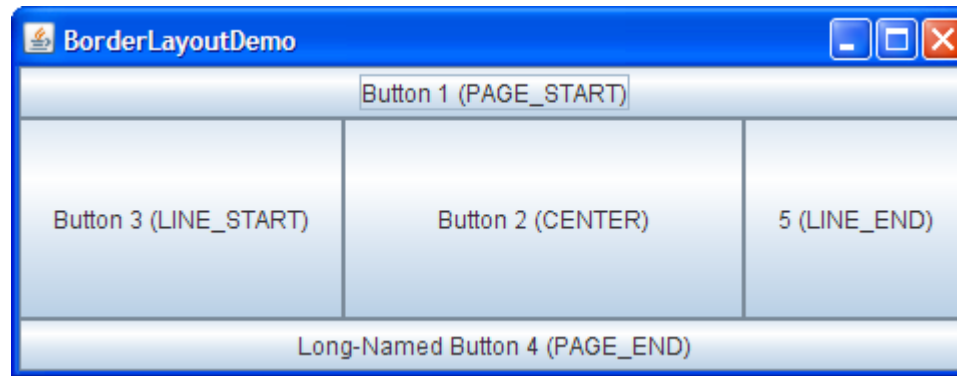
```
myFrame.setVisible(true);
```

Adding the JLabel  
component to the window

# Layout Managers

---

- A `LayoutManager` takes care of positioning sub-components in a particular way
  - `BorderLayout`, `FlowLayout`, `GridLayout` etc.
- The `BorderLayout`, for example, divides the component into five areas: NORTH, EAST, SOUTH, WEST, and CENTER



```
myFrame.add(myLabel, BorderLayout.CENTER);
```



# Layout Managers

---

- A `BoxLayout` lays components out either horizontally or vertically

```
JFrame myFrame = new JFrame("This is my window");
myFrame.setSize(1024, 768);
myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
myFrame.getContentPane().setLayout(new BoxLayout(
myFrame.getContentPane(), BoxLayout.PAGE_AXIS));
myFrame.add(new JLabel("Label 1"));
myFrame.add(Box.createVerticalGlue());
myFrame.add(new JLabel("Label 2"));
myFrame.setVisible(true);
```

# Layout Managers

---

- There are many other kinds of layout managers:
  - CardLayout
  - GridBagLayout
  - GridLayout
  - FlowLayout
  - GroupLayout
  - SpringLayout

# Buttons

---

- Buttons are created using the `JButton` class
  - They can display text and/or an icon
    - We can set the alignment of the text/icon
  - They can be enabled and disabled

```
JButton myButton = new JButton("A Button");  
myButton.setVerticalTextPosition(AbstractButton.BOTTOM);  
myButton.setHorizontalTextPosition(AbstractButton.CENTER);  
myButton.setEnabled(false);
```

# Buttons

---

- We can make buttons do something by adding an object that implements the `ActionListener` interface
  - When the button is pressed, the `actionPerformed` method of the object is invoked

```
JButton myButton = new JButton("Click Me");  
myButton.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        // Respond to the button click  
    }  
});
```

# Buttons

---

- We can also associate an action command string with the button
  - This is available via the `ActionEvent` parameter
  - Can be useful for handling multiple buttons using a single `ActionListener` object

```
JButton myButton = new JButton("Click Me");
myButton.setActionCommand("Click");
myButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println(e.getActionCommand());
    }
});
```

## Putting it together...

---

- Let's look at an example which brings all these features together
  - An application with a button and a label
    - When we press the button, the label text will change

```
public class SimpleDemo implements Runnable {
    private JLabel theLabel;
    public void run () {
        ...
    }
    public static void main(String[] args){
        javax.swing.SwingUtilities.invokeLater(new SimpleDemo());
    }
}
```

## Putting it together...

---

```
public void run() {  
    JFrame myFrame = new JFrame("Our Simple Demo");  
    myFrame.setSize(512, 384);  
    myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    theLabel = new JLabel(); // setting the field: a label with  
                             no text  
    myFrame.add(theLabel, BorderLayout.CENTER);  
    JButton theButton = new JButton("Click Me");  
    theButton.addActionListener(new ActionListener() {  
        public void actionPerformed(ActionEvent e) {  
            SimpleDemo.this.theLabel.setText("We pressed the button!");  
        }  
    });  
  
    myFrame.add(theButton, BorderLayout.SOUTH);  
    myFrame.setVisible(true);  
}
```

A reference to the instance of the class that contains the anonymous inner class object

# Introducing Threads

---

- Q: What would happen if we were to run a very time-consuming calculation in our button's action listener?

```
theButton.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        BigInteger i = ...  
        // Calculate a prime number with 10,000,000  
        digits  
        theLabel.setText(i);  
    }  
});
```

- A: Our GUI would freeze up!



# Threads

---

- The Thread class is the programming abstraction that Java uses to represent a thread of execution
- We can create new threads by:
  - Subclassing the Thread class
  - Instantiating a new Thread object using a Runnable
- Threads are started by calling the start method
- We can make the current thread wait until another thread has finished executing by calling the join method
- The static sleep method makes the current thread suspend for a set period of time (given in milliseconds)

# Threads

---

- What we could do instead is spawn a new thread to perform the calculation

```
theButton.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        Thread t = new Thread(new Runnable() {  
            public void run() {  
                BigInteger i = ... // Calculate prime number  
                theLabel.setText(i);  
            }  
        });  
        t.start();  
    }  
});
```

- The actionPerformed method will return immediately (and our GUI will remain responsive) and the thread will run concurrently

## Client/server networking (cont'd)

---

- Server uses **ServerSocket** object to listen for clients

```
// listen for client
ServerSocket ss = new ServerSocket(port);
Socket s = ss.accept();           // waits for connection

// receive data
InputStream is = s.getInputStream();
InputStreamReader isr = new InputStreamReader(is);
BufferedReader br = new BufferedReader(isr);
input = br.readLine();
```

# Networking in Java

---

- The `java.net` package contains classes for performing networking functions
- `URL`, provides a flexible high level abstraction for accessing resources over a network
- There are also classes for doing more low-level network access, e.g. TCP/UDP sockets
  - Client uses `Socket` class to connect and communicate with server
  - Server uses `ServerSocket` to listen for clients

## Client/server networking

---

- ◆ Client uses **Socket** object to connect to server

```
// connect to server
```

```
Socket s = new Socket(serverAddress, port);
```

```
// send data
```

```
OutputStream os = s.getOutputStream();
```

```
PrintStream ps = new PrintStream(os);
```

```
ps.println("Hi there!");
```