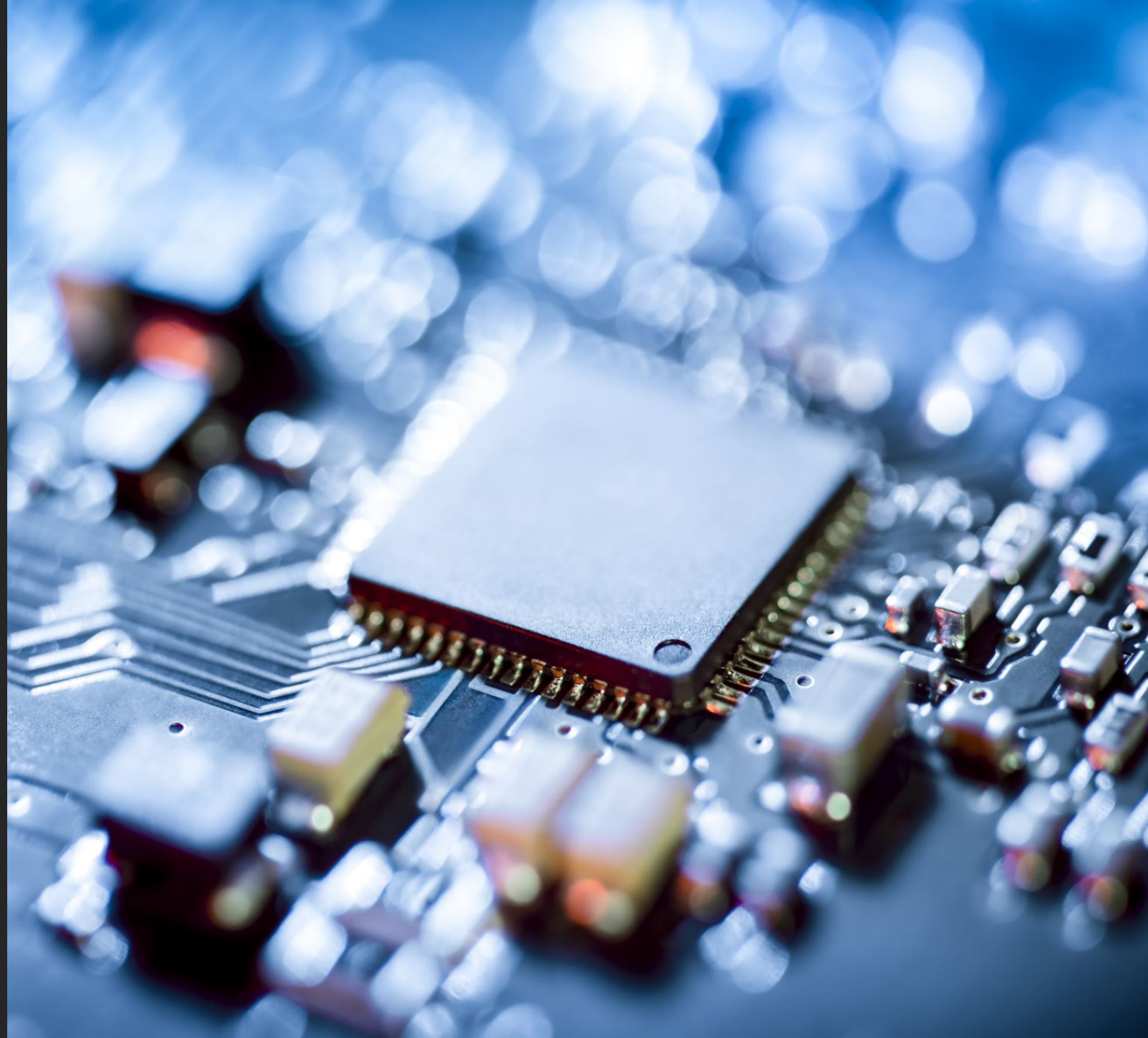
SOFTWARE ENGINEERING

ECE 651

SPRING 2020

JAVA FOR C++ PROGRAMMERS



A TALE OF TWO LANGUAGES

C w/Classes
1979

C++
1983

JAVA
1996

Better Design Decisions?

S.O.L.I.D.
2000

JAVA: LOOKS A LOT LIKE C++

```java
public class Point {
  private final int x;
  private final int y;

  public Point() {
    this(0, 0);
  }

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }

  public double distanceTo(Point otherPoint) {
    int dx = otherPoint.getX() - x;
    int dy = otherPoint.getY() - y;
    return Math.sqrt(dx * dx + dy * dy);
  }
}
```

We are making a class called “Point”
Different from C++: the class itself is public

The class has fields x and y of type **int**
These fields are private (just like C++)
final basically means “**const**”

The default constructor is public
It uses **constructor chaining** to call the 2 argument constructor with (0,0)

This constructor takes values for x and y
It initializes this.x and this.y from those values.
There are no initializer lists in Java, just assignments
In the constructor

Methods to get x and y. Note no “const” on methods Java (in C++ these would be const methods).

JAVA: LOOKS A LOT LIKE C++

```java
public class Point {
  private final int x;
  private final int y;

  public Point() {
    this(0, 0);
  }

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }

  public double distanceTo(Point otherPoint) {
    int dx = otherPoint.getX() - x;
    int dy = otherPoint.getY() - y;
    return Math.sqrt(dx * dx + dy * dy);
  }
}
```

A method that takes another Point
- Note: behaves like Point * in C++
- Any object type is always a pointer

Java only has dot (.), not arrow (->).
Always access fields and methods with dot.

sqrt is a static method in class Math.
In Java everything is inside of some class.

RUNNING OUR CODE

```java
public class Point {
  private final int x;
  private final int y;

  public Point() {
    this(0, 0);
  }

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }

  public double distanceTo(Point otherPoint) {
    int dx = otherPoint.getX() - x;
    int dy = otherPoint.getY() - y;
    return Math.sqrt(dx * dx + dy * dy);
  }

  public static void main(String[] args) {
    Point p1 = new Point(3, 4);
    Point p2 = new Point(-1, 2);
    System.out.println("p1 is " + p1);
    System.out.println("p2 is " + p2);
  }
}
```

Can add a main method to make class runnable.
- public (same meaning as C++)
- static (same meaning as C++)
- return type is void (unlike main in C++)
- takes an array of strings (String[]) as parameter
  - These are the command line arguments

String: built in class in Java
https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

Arrays:
Indexed just like C++: `args[i]`
- but perform bounds checking

Have a `length` field to say how many elements

`new Point` works much like C++:
creates a Point object
initializes it with the constructor

`System.out.println` prints its argument.
println: adds a newline
print: does not add a newline

RUNNING OUR CODE

```java
public class Point {
  private final int x;
  private final int y;

  public Point() {
    this(0, 0);
  }

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }

  public double distanceTo(Point otherPoint) {
    int dx = otherPoint.getX() - x;
    int dy = otherPoint.getY() - y;
    return Math.sqrt(dx * dx + dy * dy);
  }

  public static void main(String[] args) {
    Point p1 = new Point(3, 4);
    Point p2 = new Point(-1, 2);
    System.out.println("p1 is " + p1);
    System.out.println("p2 is " + p2);
  }
}
```

Compile:

javac Point.java

Run:

java Point

Output:

```
p1 is Point@3764951d
p2 is Point@4b1210ee
```

What went wrong?

Printing an object uses its toString() method.
Point inherits toString() from Object.
- All classes extend Object if no other parent class specified

Object's toString() gives "type@memory address".

OVERRIDE TOSTRING

```java
public class Point {
  private final int x;
  private final int y;

  public Point() {
    this(0, 0);
  }

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }

  public double distanceTo(Point otherPoint) {
    int dx = otherPoint.getX() - x;
    int dy = otherPoint.getY() - y;
    return Math.sqrt(dx * dx + dy * dy);
  }

  @Override
  public String toString() {
    return "(" + x + ", " + y + ")";
  }
  public static void main(String[] args) {
    Point p1 = new Point(3, 4);
    Point p2 = new Point(-1, 2);
    System.out.println("p1 is " + p1);
    System.out.println("p2 is " + p2);
```

You should generally write a toString() method:
Specifies how to convert an object to a String (e.g., for printing)

@Override annotation
- Specifies that this method overrides something from parent class
- Compiler gives error if not actually overriding parent (helps avoid mistakes!)

In Java, the + operator is overloaded to do String concatenation.
- Note: you cannot overload operators yourself as you can in C++.

OBJECTS VS PRIMITIVES

- In Java, Object is the parent type of all objects.
  - Its **fully qualified name** is **java.lang.Object**.
    - Fully qualified name: includes the package.
      - Packages starting with “java.” are part of the core of java. Most common are java.lang. and java.util.
      - java.lang is the most core: no need to import it.
  - Documentation https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html
    - Oracle has docs for Java library: use them!
- Java also has 8 primitive types:
  - boolean, byte, char, double, int, float, long, short.
  - Each has a “wrapper” class Boolean, Byte, Character, Double, Integer, Float, Long, Short
- Java also has “void”
  - Indicates the absence of a value. Typically as the return type

SHOULD OVERRIDE HASHCODE AND EQUALS

```
28  @Override
29  public int hashCode(){
30    return x*943 + y;
31  }
32  @Override
33  public boolean equals(Object other){
34    if (other != null && other.getClass().equals(getClass())) {
35      Point otherPoint = (Point) other;
36      return x == otherPoint.getX() && y == otherPoint.getY();
37    }
38    return false;
39  }
```

public int hashCode():

How to hash these objects when putting in hashtables (e.g. HashMap, HashSet,…).

Implementation in Object: memory address of the object

SHOULD OVERRIDE HASHCODE AND EQUALS

```
28  @Override
29  public int hashCode(){
30    return x*943 + y;
31  }
32  @Override
33  public boolean equals(Object other){
34    if (other != null && other.getClass().equals(getClass())) {
35      Point otherPoint = (Point) other;
36      return x == otherPoint.getX() && y == otherPoint.getY();
37    }
38    return false;
39  }
```

public boolean equals(Object other)

How to tell if two objects are the same:

Implementation in Object:

return this == other; //compare if pointers have same values

A FEW PARTS OF THIS

getClass():
A method in Object which returns the Class of the object
Class is a class in Java!
This checks if the `other` is the same class as `this`.

```
28  @Override
29  public int hashCode(){
30    return x*943 + y;
31  }
32  @Override
33  public boolean equals(Object other){
34    if (other != null && other.getClass().equals(getClass())) {
35      Point otherPoint = (Point) other;
36      return x == otherPoint.getX() && y == otherPoint.getY();
37    }
38    return false;
39  }
```

A FEW PARTS OF THIS

```
28  @Override
29  public int hashCode(){
30    return x*943 + y;
31  }
32  @Override
33  public boolean equals(Object other){
34    if (other != null && other.getClass().equals(getClass())) {
35      Point otherPoint = (Point) other;
36      return x == otherPoint.getX() && y == otherPoint.getY();
37    }
38    return false;
39  }
```

Casting: As in C++, we should do this sparingly
Always check for expected type before casting.

LET US LOOK AT EQUALS MORE

```
28  @Override
29  public int hashCode(){
30    return x*943 + y;
31  }
32  @Override
33  public boolean equals(Object other){
34    if (other != null && other.getClass().equals(getClass())) {
35      Point otherPoint = (Point) other;
36      return x == otherPoint.getX() && y == otherPoint.getY();
37    }
38    return false;
39  }
```

Two notions of equality for objects:

x == y: point to same object

x.equals(y) : equivalent objects

Point p1 = new Point(3,4);
Point p2 = new Point(3,4);

p1 == p2 //false: do not point at same object

p1.equals(p2) //true: equivalent (both are (3,4))

LET US LOOK AT EQUALS MORE

```
32  @Override
33  public boolean equals(Object other){
34    if (other != null && other.getClass().equals(getClass())) {
35      Point otherPoint = (Point) other;
36      return x == otherPoint.getX() && y == otherPoint.getY();
37    }
38    return false;
39  }
```

Java says your equals should follow certain rules:

Reflexivity: x==y implies x.equals(y)
Symmetry: x.equals(y) implies y.equals(x)
Transitivity: x.equals(y) and y.equals(z) implies x.equals(z)
Consistency: x.equals(y) should not change value unless we change something about the state of x and/or y inbetween
Not equal to NULL: x.equals(null) should be false

LET US LOOK AT EQUALS MORE

instanceof operator:
checks if dynamic type of other is Point, or any subclass of Point

Safe to cast to Point, but makes a broken equals method

```
32  @Override
33  public boolean equals(Object other){
34    if (other != null && other.getClass().equals(getClass())) {
35      Point otherPoint = (Point) other;
36      return x == otherPoint.getX() && y == otherPoint.getY();
37    }
38    return false;
39  }
```

A slightly different (and **incorrect**) equals:

```
@Override
public boolean equals(Object other) {
  if (other instanceof Point) {
    Point otherPoint = (Point) other;
    return x == otherPoint.getX() && y == otherPoint.getY();
  } else {
    return false;
  }
}
```

SUPPOSE WE MAKE POINT3D

Inheritance in Java.
Just like
class Point3D: public Point {
In C++

Point has this (broken) equals:

```java
@Override
public boolean equals(Object other) {
  if (other instanceof Point) {
    Point otherPoint = (Point) other;
    return x == otherPoint.getX() && y == otherPoint.getY();
  } else {
    return false;
  }
}
```

Passes argument to parent class (Point) constructor

super.equals:
Call equals inherited from parent (Point)

```java
class Point3D extends Point {
  private final int z;

  public Point3D() {
    this(0, 0, 0);
  }

  public Point3D(int x, int y, int z) {
    super(x, y);
    this.z = z;
  }

  public int getZ() {
    return z;
  }

  @Override
  public String toString() {
    return "(" + getX() + ", " + getY() + ", " + getZ() + ")";
  }
  @Override
  public boolean equals(Object other) {
    if (super.equals(other) && other instanceof Point3D) {
      Point3D otherPoint = (Point3D) other;
      return z == otherPoint.getZ();
    }
    return false;
  }

}
```

SUPPOSE WE MAKE POINT3D

Point has this equals:

```java
@Override
public boolean equals(Object other) {
  if (other instanceof Point) {
    Point otherPoint = (Point) other;
    return x == otherPoint.getX() && y == otherPoint.getY();
  } else {
    return false;
  }
}
```

Point p1 = new Point(3, 4);
Point3D p2 = new Point3D(3, 4, 7);

What is `p1.equals(p2)` ? true
What is `p2.equals(p1)` ? false

Does not obey symmetry!

```java
class Point3D extends Point {
  private final int z;

  public Point3D() {
    this(0, 0, 0);
  }

  public Point3D(int x, int y, int z) {
    super(x, y);
    this.z = z;
  }

  public int getZ() {
    return z;
  }

  @Override
  public String toString() {
    return "(" + getX() + ", " + getY() + ", " + getZ() + ")";
  }
  @Override
  public boolean equals(Object other) {
    if (super.equals(other) && other instanceof Point3D) {
      Point3D otherPoint = (Point3D) other;
      return z == otherPoint.getZ();
    }
    return false;
  }

}
```

SUPPOSE WE MAKE POINT3D

Point has this equals:

```
public boolean equals(Object other) {
  if (other != null && other.getClass().equals(getClass())) {
    Point otherPoint = (Point) other;
    return x == otherPoint.getX() && y == otherPoint.getY();
  } else {
    return false;
  }
}
```

```
class Point3D extends Point {
  private final int z;

  public Point3D() {
    this(0, 0, 0);
  }

  public Point3D(int x, int y, int z) {
    super(x, y);
    this.z = z;
  }

  public int getZ() {
    return z;
  }

  @Override
  public String toString() {
    return "(" + getX() + ", " + getY() + ", " + getZ() + ")";
  }
  @Override
  public boolean equals(Object other) {
    if (super.equals(other) && other.getClass().equals(getClass())) {
      Point3D otherPoint = (Point3D) other;
      return z == otherPoint.getZ();
    }
    return false;
  }

}
```

Point p1 = new Point(3, 4);
Point3D p2 = new Point3D(3, 4, 7);

Both correct now!

What is `p1.equals(p2)` ? false
What is `p2.equals(p1)` ? false

WARNING: SOME EDITORS DO THIS WRONG (OR LET YOU)

- Emacs will do it right: lsp-java-generate-equals-and-hash-code
  - Will do equals and hashcode together
- Some editors will do it wrong (using instanceof)
  - Or let you do it (have an option to do it incorrectly----why???)
- Of course, using Emacs is the best option ☺

- Note: some advanced situations where we want to do equality yet a different way
  - We won't talk about those today

SUMMARY SO FAR

- Java is really similar to C++
  - Same: Classes, public/private/protected, return, braces, constructors, fields, methods, int, double…
  - Small changes: inheritance, final vs const
  - Same but not shown: if, while, for
  - Differences:
    - Only pointers to object types (but no explicit *)
    - instanceof
    - All classes extend Object (or some subclass of it) and should override certain methods
    - @Override annotation
  - Some methods we typically need to write (override Object's implementation)
    - public String toString(): how to convert object to a string, e.g., for printing
    - public boolean equals(Object other) : test if two objects are the same
      - use getClass() to check for same type!
    - public int hashCode() : how to hash for hashtables

JAVA CLASSES FOR C++ PROGRAMMERS

C++ Class	Java Class	Notes
std::vector<T>	java.util.ArrayList<T>	java.util.Vector<T> is just like array list, except thread safe
std::list<T>	java.util.LinkedList<T>	
std::string	java.lang.String	
std::map<K,V>	java.util.HashMap<K,V>	Or java.util.TreeMap<K,V> (1)
std::set<T>	java.util.HashSet<T>	Or java.util.TreeSet<T> (1)
std::iterator<T>	java.util.Iterator<T>	Similar: java.util.Enumeration<T>
std::istream	java.io.InputStream or java.io.Reader	
std::ostream	java.io.OutputStream or java.io.Writer	
std::ifstream	java.io.FileReader	Wrap in java.io.BufferedReader for lines of text
std::ofstream	java.io.PrintWriter	
std::stringstream	java.io.ByteArrayInputStream	For reading from a String
std::stringstream	java.io.ByteArrayOutputStream	For building up a String as output

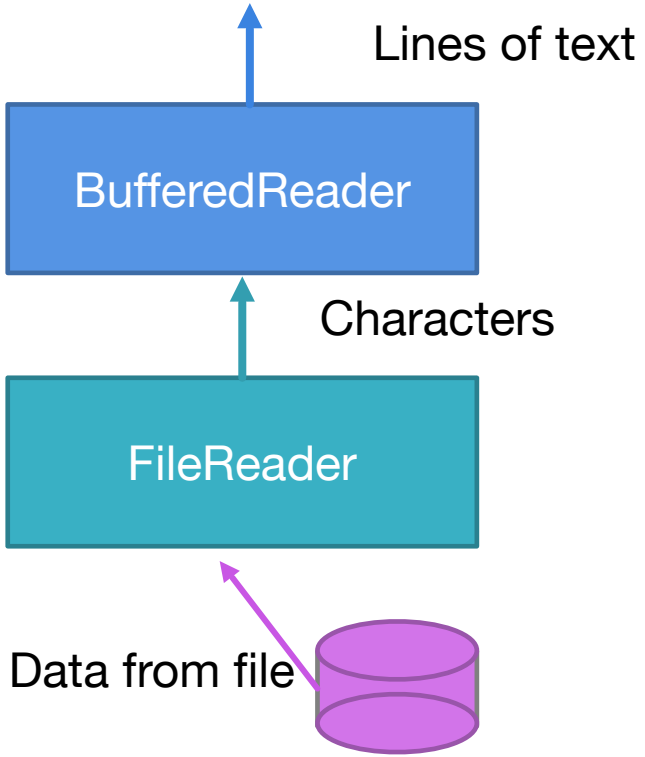
(1) C++'s map/set are implemented with balanced trees, so TreeMap/TreeSet are most similar. However, HashMap/HashSet are most common

A FEW WORDS ABOUT JAVA IO

- Java distinguishes between characters (Unicode) and bytes (raw data)
  - Different classes to read/write them
  - Streams: operate on bytes
    - E.g., InputStream, OutputStream
  - Readers/Writers: operate on characters
    - E.g., InputReader, BufferedReader, PrintWriter

A FEW WORDS ABOUT JAVA IO

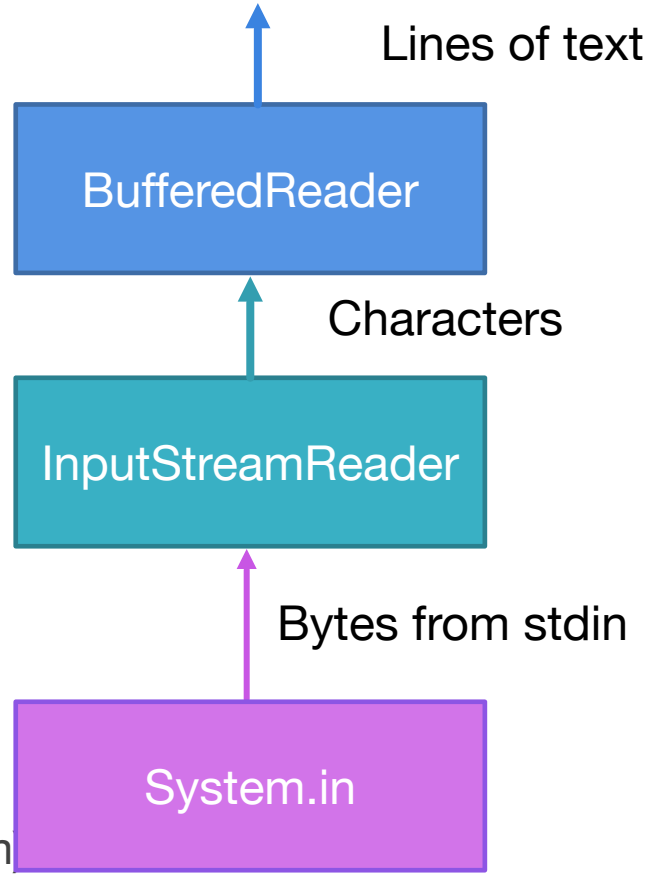
```
FileReader reader = new FileReader("myFile.txt");
BufferedReader br =new BufferedReader(reader);
```



- Java's IO library is built on the idea of wrapping a sophisticated type with a simpler type to build up functionality. Example:
  - **FileReader**: get characters out of a file
  - **BufferedReader**: get lines of text out of any other reader

A FEW WORDS ABOUT JAVA IO

```
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br =new BufferedReader(isr);
```

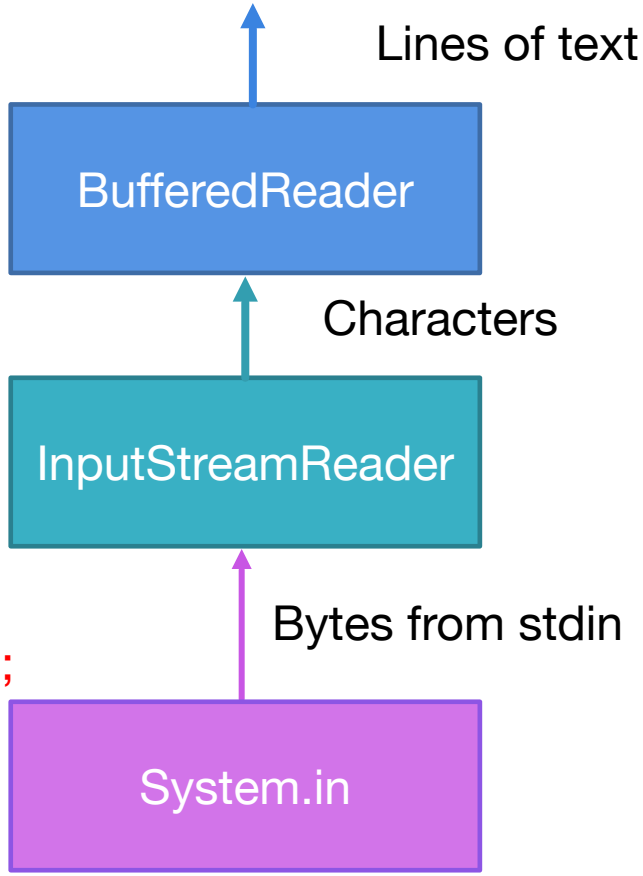


- Another example: reading lines from System.in (which is an InputStream
  - **InputStream**: provide bytes read from some source
  - **InputStreamReader**: read bytes from an input stream, convert them to characters
  - **BufferedReader**: read lines of text from any other reader

AVOID PITFALL: ONLY WRAP ONCE!

```
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br =new BufferedReader(isr);

//later…
//do not do this!
InputStreamReader isr2 = new InputStreamReader(System.in);
BufferedReader br2 =new BufferedReader(isr2);
```



- Do not wrap the same stream twice.
  - Instead, wrap it once and use the wrapped stream (e.g., “br”) throughout the program
- The first wrapping may read more data than you want it to
  - Making that data unavailable at the second wrapping
  - Often will result in very different behavior with e.g., standard in redirected from file vs not.

OBJECT STREAMS

- Java has ObjectInputStream and ObjectOutputStream
  - Designed for object **serialization**: writing object graphs to a file
  - Objects that use them need to implement java.io.Serializable interface
    - Does not require any methods, just says “this type may be serialized”

OBJECT STREAMS

- Java has ObjectInputStream and ObjectOutputStream
  - Designed for object **serialization**: writing object graphs to a file
  - Objects that use them need to implement java.io.Serializable interface
    - Does not require any methods, just says “this type may be serialized”
- Warning: writing same object twice does not update
  - Write the same object twice? Will remember that it is same object, reading object back will just give reference to first object written
    - Past students: I wrote this object again and none of its state got updated!
    - Can explicitly .reset() to make it “forget” past objects
- Also note that format is not human readable
  - Hard to debug (am I sending right values?)
  - Might consider JSON serialization with e.g., Jackson ObjectMapper

THINK, PAIR, SHARES (COMING UP)

I will introduce a difference between C++ and Java, you will think about design principles that the difference might support.

QUICK REMINDER – OO DESIGN PRINCIPLES

- Abstraction, Encapsulation, Inheritance, Polymorphism!
- Effective parallelism between developers requires independent tasks
- Least Surprise
- DRY: Don't Repeat Yourself
- Low Coupling / High Cohesion
- SOLID
- Design for testability

METHOD DISPATCH

C++

- Can request dynamic dispatch
  - Calls to an overridden method is resolved at runtime
- How do we request it?
  - **virtual** on declaration in class of static type (or its parents)

Java

- Dynamic dispatch for every method
  - No other choice

DESIGN PRINCIPLES ADDRESSED

- **Least Surprise**: Might expect dynamic dispatch
  - Especially if you wrote virtual in the child class
- **Open/Close**: Didn't make it virtual to begin with? Need to modify parent
  - What if we we had a different subclass with same method/static dispatch?

MEMORY ALLOCATION FOR OBJECTS

C++

- Objects created in Heap or Stack
- Object in the **frame**
  - Destroyed when function returns
- Heap management
  - The memory containing the object persists until the end of your program, or until you delete the object
  - Uses delete + destructors

JAVA

- All Objects created in the Heap
- No such thing as objects in the **frame**
- Heap management
  - Garbage collection – freeing any object without reference in the method
- No destructors

DESIGN PRINCIPLES ADDRESSED

- **Least surprise**: no strange bugs from free-related errors
- **Better abstraction**: don’t need to know where memory is allocated or when to free

NO OBJECTS IN THE FRAME: WHAT'S THE CONSEQUENCE?

- No Resource Acquisition is Initialization (RAII)
- What is RAII?
  - Local object owns resource, responsible for destruction
- How do we handle non-memory resources?
  - Brainstorm: what might we need to deal with other than memory?

NON-MEMORY RESOURCES

- Files:
  - Need to close the file
  - What if an exception is thrown between open and close?

- Locks:
  - Need to unlock a lock if you lock it
  - What happens if an exception happens in **critical section**?

MULTI-THREADING QUICK REVIEW

- Multiple threads execute concurrently
- Goal: perform different work in parallel -> speed up code
- Difficulty: may need to access same data
  - Need to prevent **race conditions**: where thread interleaving affects correctness of code
  - Protect **critical sections** with a mutex (aka “lock”)
    - **Critical section**: region of code that must have at most one thread at any time
    - **Mutex**: synchronization construct that can only **be acquired (locked)** by one thread at a time
    - Threads must **unlock** the mutex when leaving the critical section.

In C:

```
pthread_mutex_lock(&m);
//critical section
pthread_mutex_unlock(&m);
```

What if an exception happens here?

C++ USES RAII

- C++11 uses RAII for mutexes with std::lock_guard<std::mutex>
  - Constructor takes a std::mutex and locks it
  - Destructor unlocks the mutex
- Other variants for locks who may have ownership transferred (std::unique_lock) etc.

- But Java does not have RAII…

JAVA LOCKS

- Java builds locks into the language directly including a keyword for critical sections:
  - class java.lang.Object has a built in mutex
    - It also has a condition variable and wait/notify/notifyAll methods (like pthread_cond_wait, pthread_cond_signal, pthread_cond_broadcast).
    - Every class inherits from Object
- The keyword synchronized makes a critical section,

```
synchronized(object) {
  //critical section code
}
```

  - The object here is the object whose mutex should be locked (commonly: **synchronized**(**this**) {…}).
  - Mutex is unlocked whenever the block is left, even if by an exception.
  - Note: synchronized can also be put in a method declaration. Just like wrapping entire method body in synchronized(this){…}

JAVA LOCKS

```
import java.util.LinkedList;
public class SyncQueue<T> {
  private LinkedList<T> myData;
  public SyncQueue() {
    myData = new LinkedList<>();
  }
  public synchronized void add(T toAdd) {
    myData.add(toAdd);
    notify();
  }
  public synchronized T get() throws InterruptedException {
    while(myData.size() == 0) {
      wait();
    }
    return myData.remove();
  }
}
```

Note the <T>
This is a Java Generic---provides parametric Polymorphism (like C++ templates).
Some differences though---will talk about soon

This method is **synchronized**:
It locks this object at the start and unlocks at end. Same as **synchronized**(**this**){…}

This method is also **synchronized**:
They lock the same mutex (in this object)

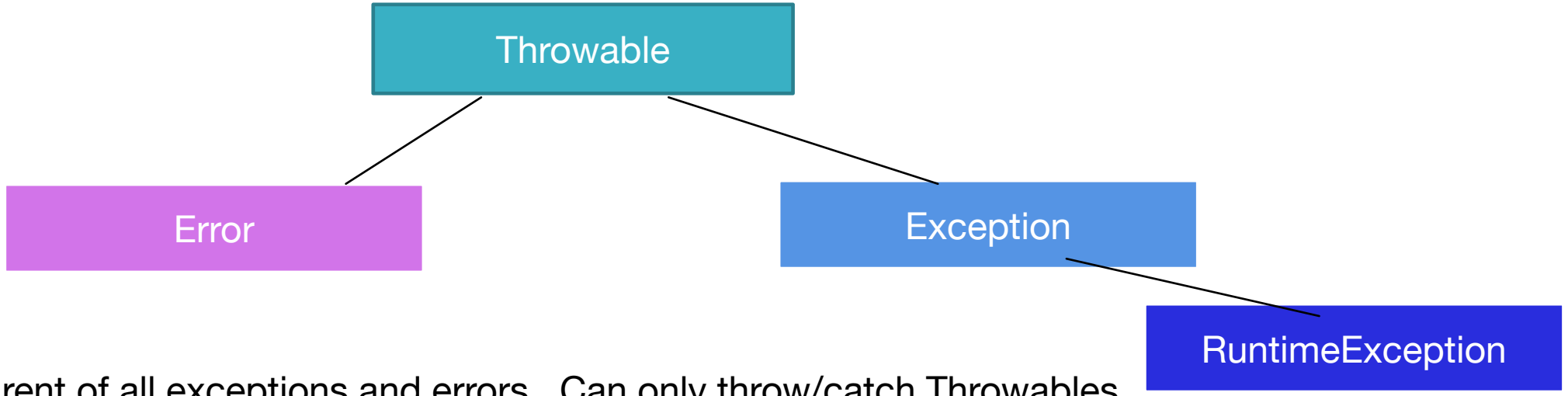
wait() and notify() called on this object too
(implicitly this.wait() and this.notify())
Work like pthread_cond_wait/pthread_cond_signal

TRY, CATCH, AND FINALLY

```
try{
    open a file
    do stuff with the file
}
catch(SomeException e) {
    //do whatever with e
}
finally {
    close the file
}
```

- Java has try-catch much like C++
  - Don't catch by reference (no explicit references in Java)
  - When throwing, "new" the exception to throw (unlike C++)
  - Java "fixes" exception specifications
    - Recall: C++03 you can declare what a method might throw. C++11 deprecates it
    - C++ checks them at runtime
    - Java has a better exception specification + checks at compile time
- Java adds **finally**
  - Code in a **finally** block is ALWAYS done (exception or not).

JAVA THROWABLES (EXCEPTIONS + ERRORS)



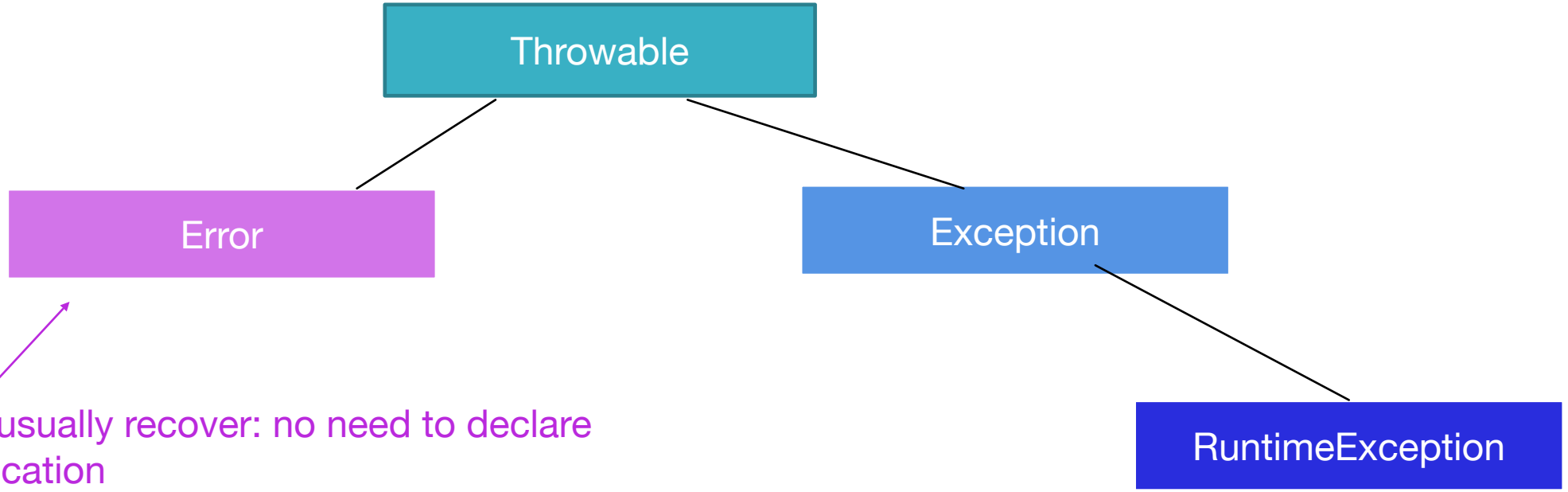
Throwable: parent of all exceptions and errors. Can only throw/catch Throwables

Error: things that are so bad they should be quite rare and are generally unrecoverable (out of memory, corrupted class file,..)

Exception: parent of exceptions. You generally throw + catch subclasses of this

RuntimeException: exceptions that could happen practically anywhere
NullPointerException: could happen at any object.method() or object.field
ArrayIndexOutOfBoundsException: could happen at any array index

JAVA THROWABLES (EXCEPTIONS + ERRORS)



Can not usually recover: no need to declare in specification

Everything else: “Checked Exception”: declare in specification if might be thrown
Checked by compiler!

Could happen anywhere: no need to declare in specification

CHECKED EXCEPTIONS

java.io

Class FileNotFoundException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.io.IOException
        java.io.FileNotFoundException

FileInputStream

```
public FileInputStream(String name)
                throws FileNotFoundException
```

Creates a `FileInputStream` by opening a connection to an actual file, the file named by the path name `name` in the file system. A new `FileDescriptor` object is created to represent this file connection.

First, if there is a security manager, its `checkRead` method is called with the `name` argument as its argument.

If the named file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading then a `FileNotFoundException` is thrown.

Parameters:

`name` - the system-dependent file name.

Throws:

`FileNotFoundException` - if the file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading.

`SecurityException` - if a security manager exists and its `checkRead` method denies read access to the file.

See Also:

`SecurityManager.checkRead(java.lang.String)`

java.lang

Class SecurityException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        java.lang.SecurityException

EXCEPTION SPECIFICATIONS (“THROWS…”)

```
public void readAndPrintFile(String name) {
  FileInputStream fis = new FileInputStream(name);
}
```

new FileInputStream(name) might throw FileNotFound
- we do not catch it
- nor do we declare it in the “throws” clause

This is an error

Emacs gives us more specific info when the cursor is on the problem

```
34
35  public void readAndPrintFile(S➤ Unhandled exception type FileNotFoundException [16777384]
36    FileInputStream fis = new FileInputStream(name);
37  }
-UUU:----F1  Point3D.java    Top (36,32)    (Java//l LSP[jdtls:23434] yas Gradle FlyC company El
java.io.FileInputStream.FileInputStream(String name) throws FileNotFoundException
```

```
public void readAndPrintFile(String name) throws FileNotFoundException {
  FileInputStream fis = new FileInputStream(name);
}
```

We can say that this method throws the exception and the caller has to deal with it (or declare that it can propagate

TRY…FINALLY

```
public void readAndPrintFile(String name) throws IOException {
  BufferedReader br = new BufferedReader(new FileReader(name));
  try {
    String s;
    while((s = br.readLine()) != null) {
      System.out.println(s);
    }
  }
  finally {
    br.close();
  }
}
```

We will improve on next slide

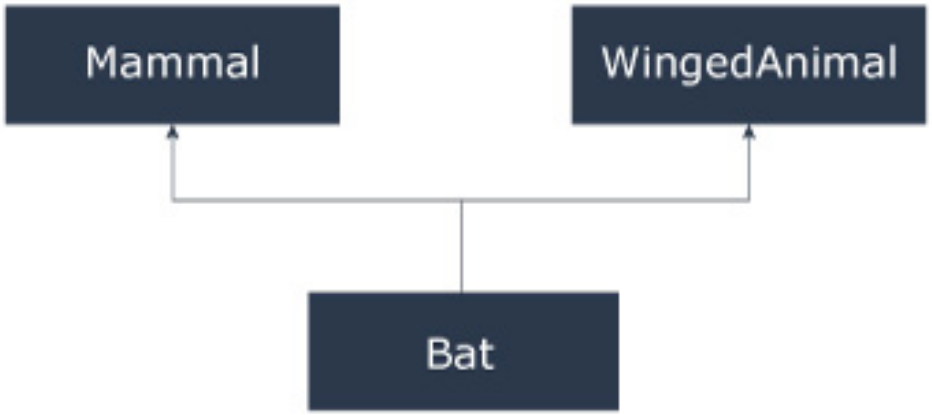
TRY WITH RESOURCE

```
public void readAndPrintFile(String name) throws IOException {
  try (BufferedReader br = new BufferedReader(new FileReader(name))) {
    String s;
    while ((s = br.readLine()) != null) {
      System.out.println(s);
    }
  } //implicit: finally{ if(br!= null) {br.close();}}
}
```

INHERITANCE

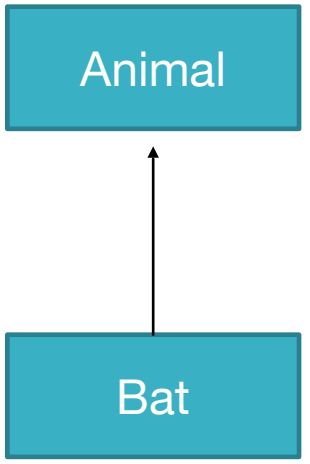
C++

- Multiple Inheritance
  - A subclass can inherit from more than one superclass



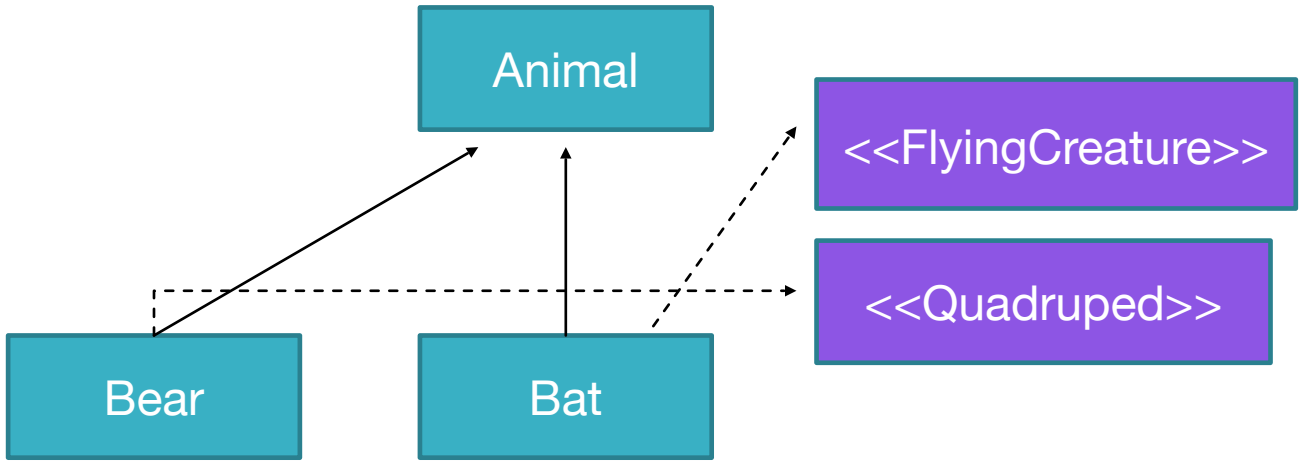
Java

- Single inheritance
- Uses interfaces instead



INTERFACES IN JAVA

- Specify what a class must do (method specification), but not how (implementation)
- No fields
  - Except static final constants



ABSTRACT CLASS VS INTERFACE

```
public interface FlyingCreature {
    public void flyTo(Location where);
}
```

```
public abstract class Animal {
    private String name;
    public Animal(String name) {
        this.name = name;
    }
    public String getName() {
      return name;
    }
    public abstract void eat(Food f);
}
```

```
public class Bird extends Animal implements FlyingCreature{
  …
}
```

- Interfaces are “purely” abstract classes: no fields
  - Typically, no method implementations (Java added “default” methods in interfaces in Java 8).
  - Static fields final fields are allowed (constants)
  - Static methods and static initializer blocks also allowed: generally a bad idea

DESIGN PRINCIPLES ADDRESSED

- **Interface Segregation Principle:** Can split interfaces without complications of multiple inheritance
- **Dependency Inversion Principle**: can depend on just an interface

PARAMETRIC POLYMORPHISM

- A programming language technique that enables the generic definition of entities (classes, functions, methods), to improve code re-use
- Entities are parameterized over one or more types (e.g., “T”)
  - Can be used for any T
  - E.g., LinkedList<int>, LinkedList<String>,…

PARAMETRIC POLYMORPHISM

C++

- Called “templates”
- Recompiled for each T it is used with
- Type checking done at use
- Code must be directly visible at use

Java

- Called “generics”
- Compiled once, re-used for all Ts
  - T is “erased”: not available at runtime
- Type checking done at definition
- Can use compiled class files normally

PARAMETRIC POLYMORPHISM

- True independence of the type (really for all) is restrictive
  - Want to order things? Not all types are orderable.
  - Want to check for equality? Not all types support equality testing.
  - Want to ….? Not all types support …

JAVA GENERICS

- Once Glass<T> is compiled works for any type
- Compiles one version of Glass for Ts it is used with
- T goes away, and is turned into Object (Type Erasure)

```
8  class Glass<T> {
9      private T liquid;
10     //other things elided
11 }
12
13 public class Brunch {
14     Meal makeBrunchSpecial() {
15         Glass<Juice> = new Glass<Juice>();
16         Juice juice = getJuiceOfTheDay();
17         //……
18     }
19 }
```

BOUNDED POLYMORPHISM

```
30 Glass<Cake> cakeGlass = new Glass<Cake>();
31 //this doesn't make sense!
```

- Could use "Cake" as a parameter, but this is not really what you want.
- Instead can restrict generic to bounded type parameters
- Now glass instantiations will only accept liquids

```
22 public interface Liquid {
23     //various methods
24 }
25 //…
26 class Glass<T extends Liquid> {
27     private T liquid;
28 }
```

THINK, PAIR, SHARE

- What design principles do generics address?

DESIGN PRINCIPLES ADDRESSED

- **Least surprise**: don't get compiler errors in a class you've used many times
- **Abstraction: clear interface**: know exactly what we need to use as type parameter
- "Parametric analog of Liskov Substitution Principle" – Drew
  - LSP basically says if S is a subtype of T, code works fine if use S where T expected
  - Parametric analog : if code parameterized over <T>, and can pass S in for that parameter, code should work.

OPERATOR OVERLOADING & USER-DEFINED CONVERSIONS

C++

- Allows operator overloading
  - E.g., overload <, ==, etc. to use Standard Template Library
- Many user-defined implicit conversions
  - One argument constructors (how do we prevent implicit use?)
  - operator type()

Java

- Allows overload parameter lists on methods
- No user-defined overloading of operators
  - Easily abused
  - Makes confusing code
- No user-defined implicit conversions
  - Good b/c not surprised by them (Least Surprise x 100)

UNIT TESTING

src/**main**/java/edu/duke/adh39/stuff/Something.java

```
class Something{
   // some code
}
```

- We are going to talk a lot more about testing soon
  - But a quick intro to unit testing in JAva for now..
  - Unit testing in Java with **JUnit**

src/**test**/java/edu/duke/adh39/stuff/SomethingTest.java

```
class SomethingTest {
   @Test
   public void test_someFun() {
    Something s = new Something(42);
    assertEquals(12, s.someFun(3));
    assertEquals(19, s.someFun(7));
   }
   @Test
   public void test_anotherFun() {
     Something s = new Something(99);
     assertEquals("hello", s.anotherFun());
   }
}
```

UNIT TESTING

src/**main**/java/edu/duke/adh39/stuff/Something.java

```
class Something{
   // some code
}
```

src/**test**/java/edu/duke/adh39/stuff/SomethingTest.java

```
class SomethingTest {
   @Test
   public void test_someFun() {
    Something s = new Something(42);
    assertEquals(12, s.someFun(3));
    assertEquals(19, s.someFun(7));
   }
   @Test
   public void test_anotherFun() {
     Something s = new Something(99);
     assertEquals("hello", s.anotherFun());
   }
}
```

- Write only a LITTLE bit of code before testing
  - One method
  - Maybe only part of a method
  - Test frequently!
  - Get 100% coverage: **make sure your tests give you confidence**!
    - Refactor code -> run tests -> pass tests -> be sure you did not break anything!

UNIT TESTING

src/**main**/java/edu/duke/adh39/stuff/Something.java

```
class Something{
   // some code
}
```

C-x t

src/**test**/java/edu/duke/adh39/stuff/SomethingTest.java

```
class SomethingTest {
   @Test
   public void test_someFun() {
    Something s = new Something(42);
    assertEquals(12, s.someFun(3));
    assertEquals(19, s.someFun(7));
   }
   @Test
   public void test_anotherFun() {
     Something s = new Something(99);
     assertEquals("hello", s.anotherFun());
   }
}
```

- Emacs setup to help you do this well
  - **C-x t** switch between main and test code
  - **C-c C-t** run tests + display coverage (uses clover)
    - Green = covered line
    - Red = uncovered line
    - Yellow = partially covered branch