CS206

I/O Methods

Files/Exceptions

Inheritance

How integers are stored

- Everything is bits
  - 0 or 1
- the int type uses 32 bits with number in base 2
- To show +/- the leftmost bit
  - "sign bit"
  - 0—positive
  - 1—negative
  - "two's complement"

Suppose you have 4 bits for a number

base 10	in bits
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
-8	1000
-7	1001

Access Control Modifiers

- `public` — all classes may access
- `private` — access only within that class.
- `protected` — access only from decendents
- "" (read as package) — access only by classes within the package
    - (I hate significant whitespace)
  - The package is generally the code you are working on.
  - packages very useful in large development projects (>10 people)
  - DO NOT use in this class

Static

- When a variable or method of a class is declared as `static`, it is associated with the class as a whole, rather than with each individual instance of that class.
- **Only acceptable use** (at least for this course):
  - In methods ...
    - `public static void main(String[] args)`
  - In variables .. to declare constants
    - `public static final double GOLDEN_MEAN =1.61803398875;`

Casting (of base types)

- Assignment REQUIRES type equality
- Use casting to change type
- Must explicitly cast if there is a possible loss of precision

```
private void trial()
    {
        int x = 5;
        double y = 1.2;
        y = x;
        x = y;

        y = (double) x;
        x = (int) y;
    }
```

Wrapper Types

- Most data structures and algorithms work only work with object types (not base types).
- To get around this obstacle, Java defines a wrapper class for each base type.
- Implicitly converting between base types and their wrapper types is known as automatic boxing and unboxing.

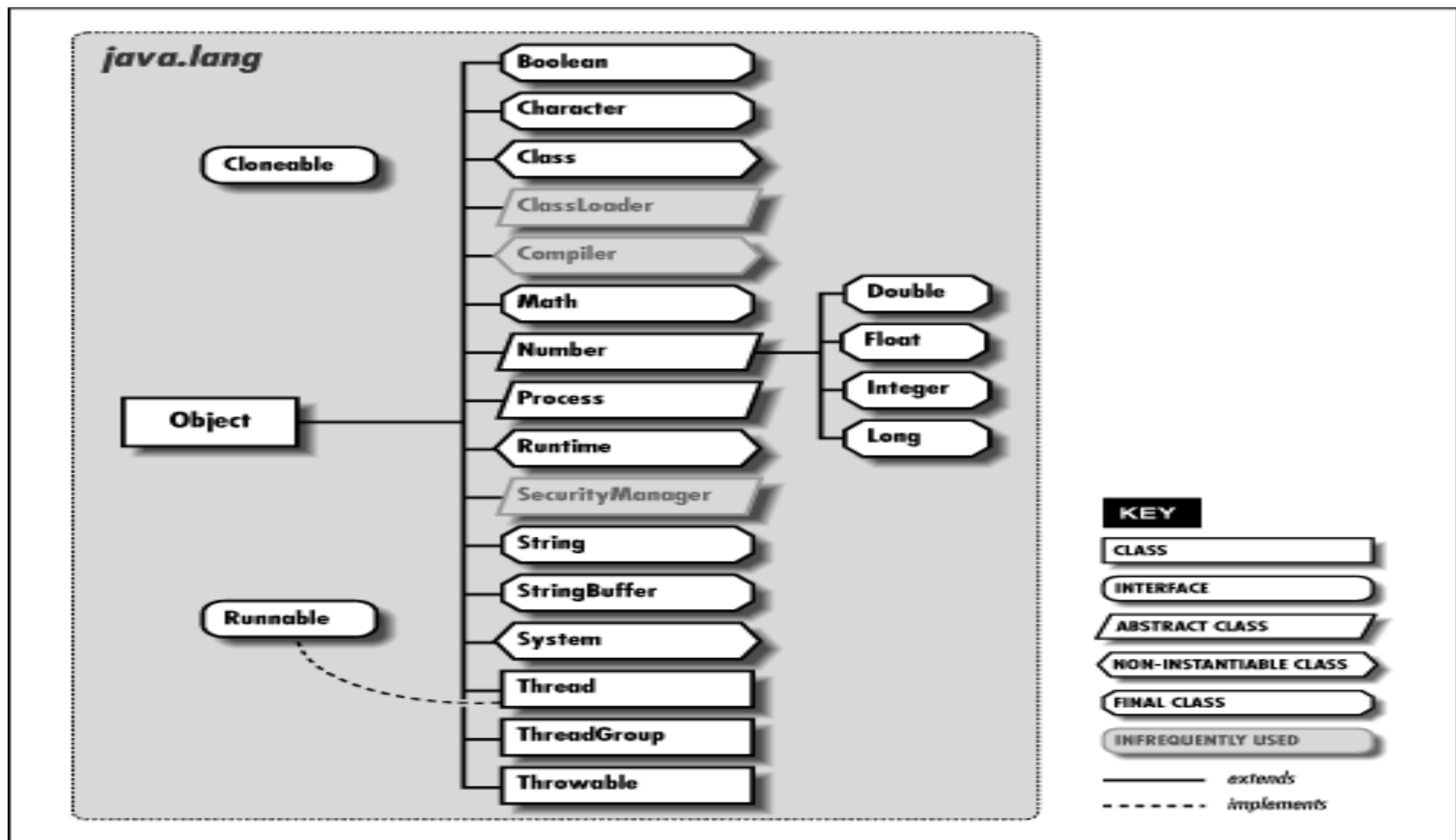
Autoboxing and unboxing

```
public class Wrapper
{
    public void w1(Integer ii) {
        System.out.println(ii);
        int i3 = ii; // auto unboxing
        System.out.println(i3*i3);
        System.out.println(i3*ii); // auto unboxing
    }
    public static void main(String[] args) {
        Wrapper w = new Wrapper();
        w.w1(5); // autoboxing
    }
}
```

What you should know/review

- variables
- expressions
- operators
- methods
  - parameters
  - return value
- conditionals
- `for`/`while` loops
- class design and object construction
  - instance variables
  - constructor
  - getters/setters
  - class methods
  - `new`
- arrays
- arrays of objects
- `String`

Start of the Java class hierarchy

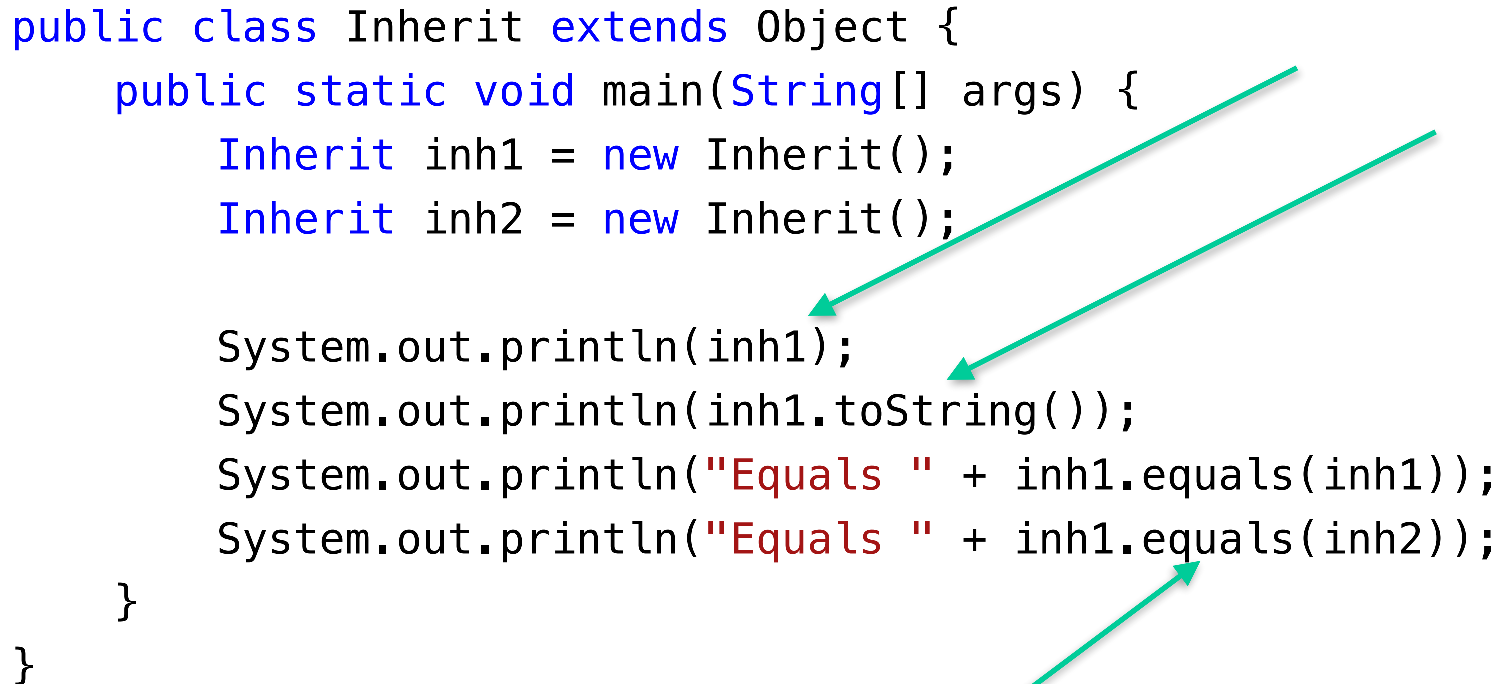


http://web.deu.edu.tr/doc/oreily/java/langref/ch10_js.htm

Java Object Methods

- **public boolean equals(Object ob)**
- **public String toString()**
- public Class getClass()
- protected Object clone()
- protected void finalize()
- public int hashCode()
- public void notify()
- public void notifyAll()
- public void wait()
- public void wait(long l)
- public void wait(long l, int ii)

Inheritance in Java



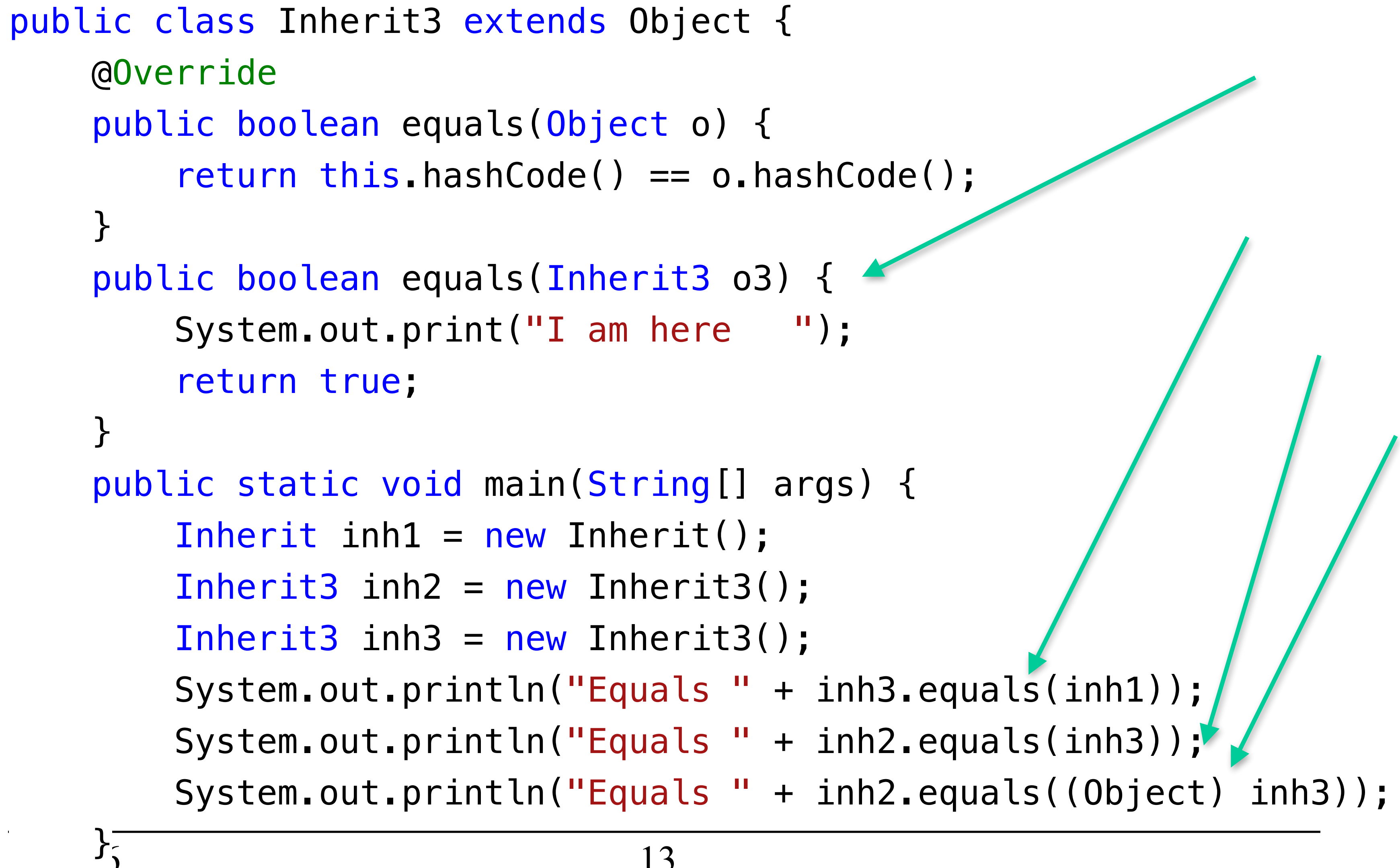
```
public class Inherit extends Object {
    public static void main(String[] args) {
        Inherit inh1 = new Inherit();
        Inherit inh2 = new Inherit();

        System.out.println(inh1);
        System.out.println(inh1.toString());
        System.out.println("Equals " + inh1.equals(inh1));
        System.out.println("Equals " + inh1.equals(inh2));
    }
}
```

Overriding Inheritance

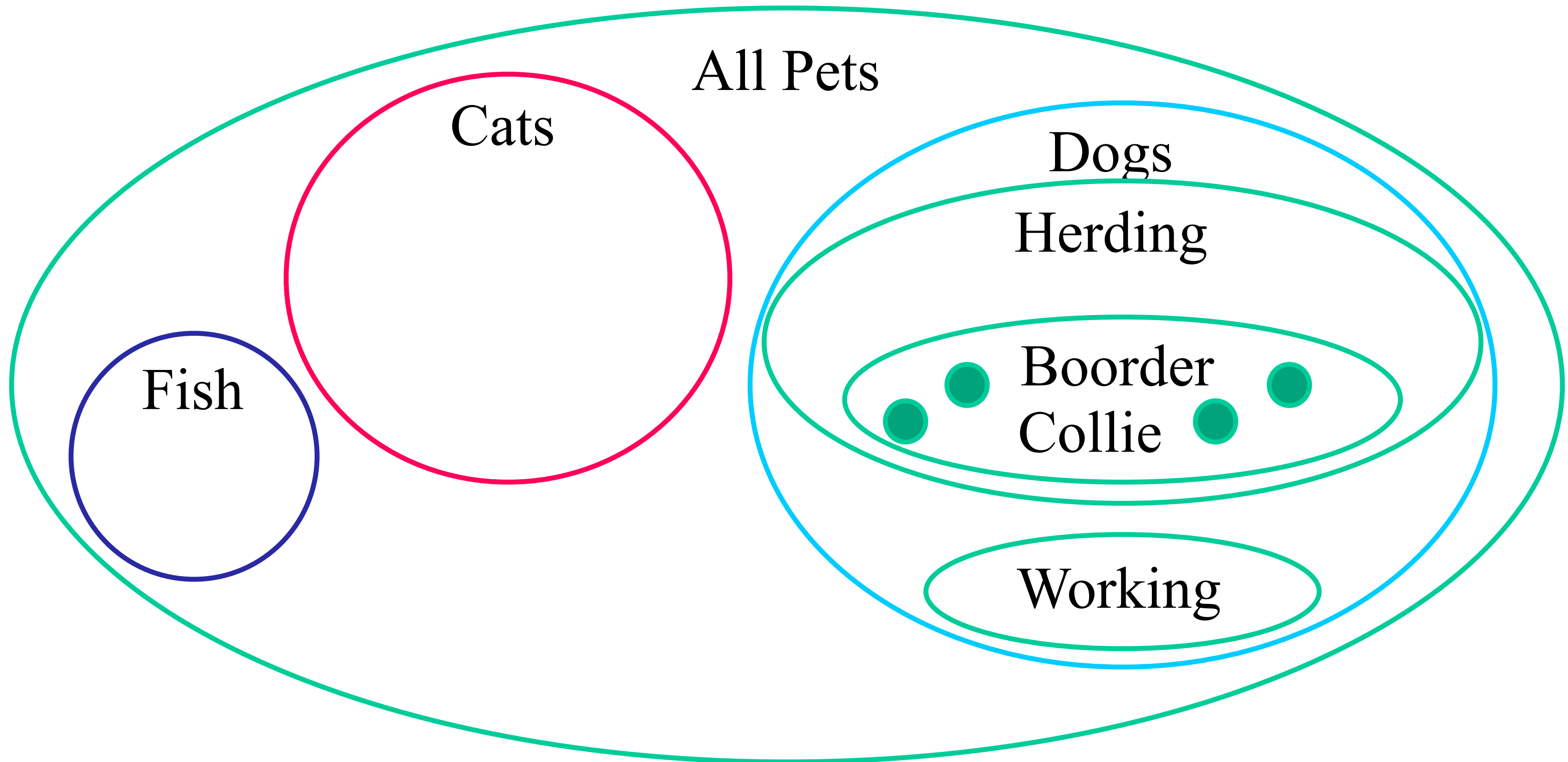
```
public class Inherit2 extends Object {
    @Override
    public String toString() {
        return "Inherit2 toString " + super.toString();
    }
    @Override
    public boolean equals(Object o) {
        return this.hashCode() == o.hashCode();
    }
    public static void main(String[] args) {
        Inherit inh1 = new Inherit();
        Inherit2 inh2 = new Inherit2();
        System.out.println(inh1);
        System.out.println(inh2);
        System.out.println("Equals " + inh1.equals(inh1));
        System.out.println("Equals " + inh2.equals(inh1));
    }
}
```

Overloading

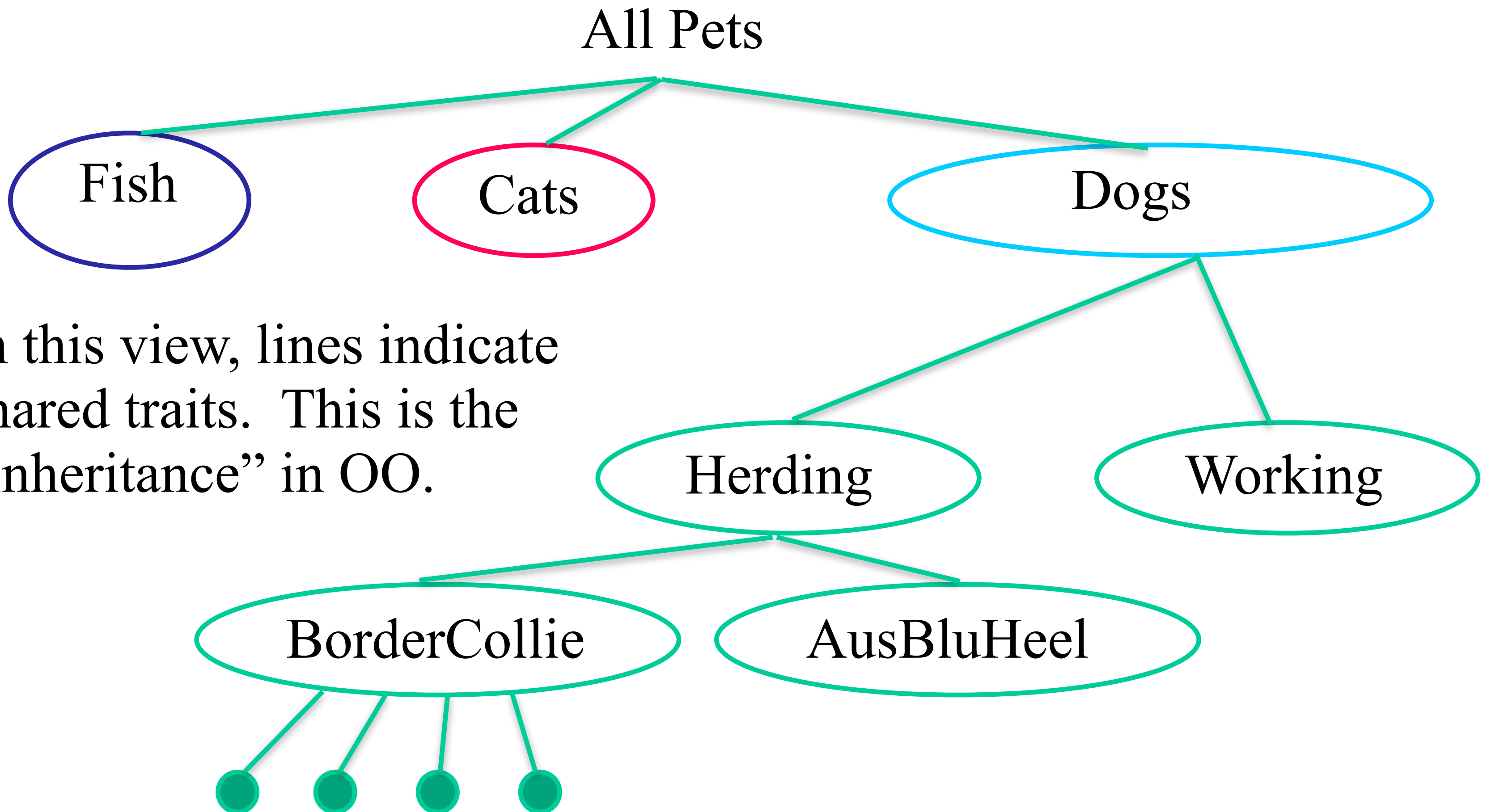


Classes and Inheritance

Consider Pets in a classic Venn Diagram view



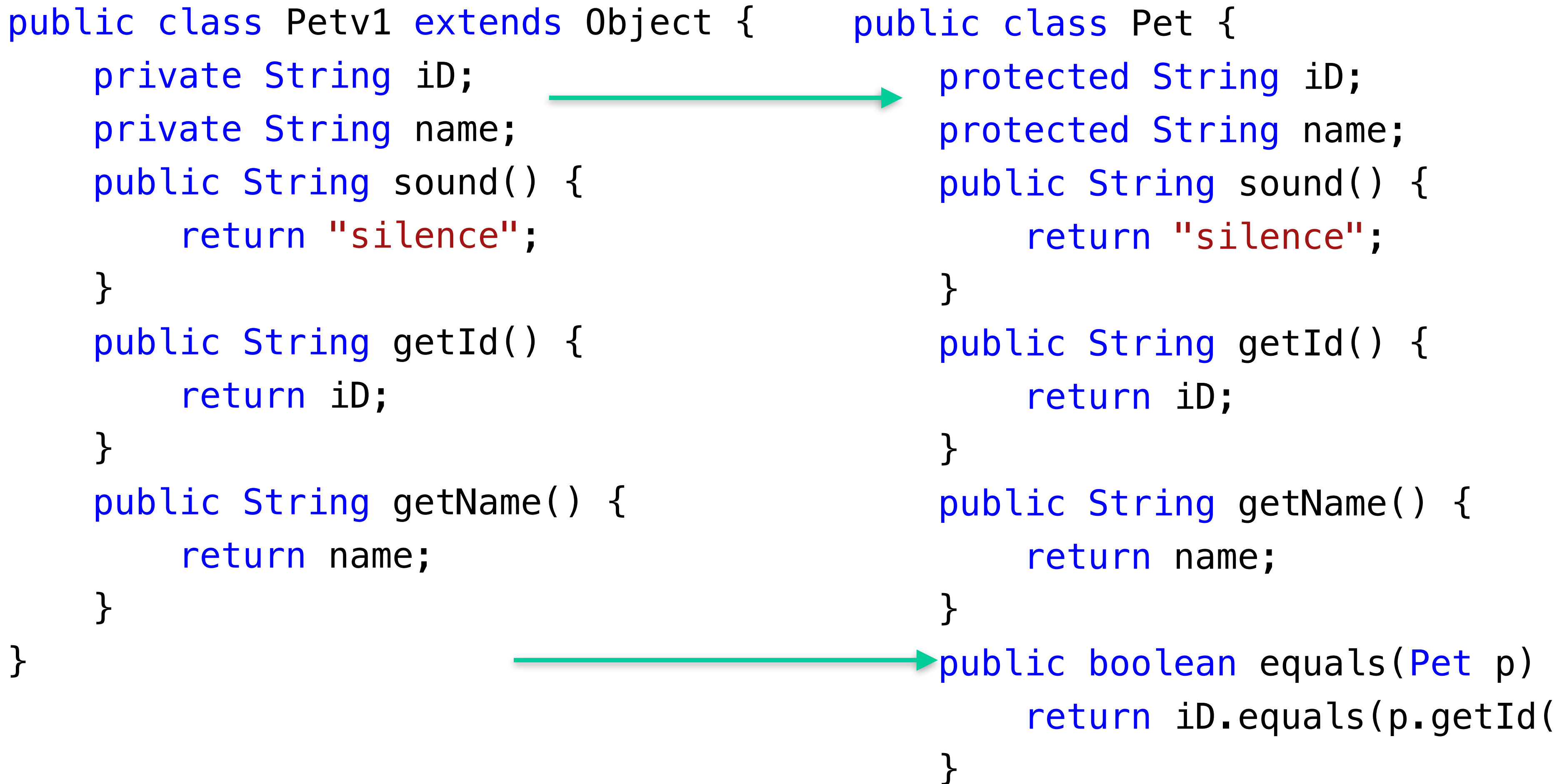
Classes and Inheritance



Classes and Inheritance

PET	CAT	DOG	WORKINGDOG
Id	Id	Id	Id
Name	Breed	Group	
Sound	Name	Breed	Breed
	Sound	Name	Name
	hair length	Sound	Sound
		hairLength	hairLength
		doubleCoat	doubleCoat
			typeOfWork

Pet Class



Cat class

```java
public class Cat extends Pet {
   private String breed;
   private double hairLength;
   public Cat(String name, String id, String breed) {
        this.name = name;
        this.iD = id;
        this.breed = breed;
    }
    @Override
    public String sound() {
        return "meow";
    }
    @Override
    public String toString() {
        return "My name is " + name + " breed " + breed + " and I say 
" + sound();
    }
    public static void main(String[] args) {
        System.out.println(new Cat("calypso", "112234", "siberian"));
    }}
```

Dog Classes

```
public class Dog extends Pet{
    protected String group;
    protected double hairLength;
    protected boolean doubleCoat;
    @Override
    public String sound() {
        return "arf";
    }
    @Override
    public String toString() {
        return sound();
    }
    public static void
main(String[] args) {
        System.out.println(new
Dog());
    }
}
```

```
public class WorkingDog extends Dog
{
    protected String breed;
    protected String task;
    @Override
    public String toString() {
        return super.toString() + "
work " + task;
        }
    @Override
    public String sound() {
        return "woof";
    }
}
```

Casting, Classes and Inheritance

- Suppose: SPCA pet shelter
- Desire: A program that tracks all animals at shelter
- Approach
  - Use single array to hold all Pets
  - But deal with dogs cats separately later

```
public class Shelter {
    Pet[] animals = new Pet[100];
    int animalCount=0;
    public void addAnimal(Pet animal) {
        animals[animalCount++]=animal;
    }
    public Pet getAnimal(int location) {
        return animals[location];
    }
   public static void main(String[] args) {
        Shelter shelter = new Shelter();
        shelter.addAnimal(new Dog());
        shelter.addAnimal(new Cat());
        Pet aa = shelter.getAnimal(1);
        System.out.println(aa);
        if (aa instanceof Cat) {
            Cat c = (Cat)aa;
            System.out.println(c.toString());
}}}
```

Exceptions

- Unexpected events during execution
  - unavailable resource
  - unexpected input
  - logical error
- In Java, exceptions are objects
- 2 options with an Exception
  - "Throw" it
    - this says that the exception must be handled elsewhere
  - "Catch" it.
    - handle the problem here and now

Catching Exceptions

- Exception handling
  - `try-catch`
- An exception is caught by having control transfer to the matching `catch` block
- If no exception occurs, all `catch` blocks are ignored

$$
\begin{array}{l}
\textbf{try}\ \{ \\
\quad \textit{guardedBody} \\
\}\ \textbf{catch}\ (\textit{exceptionType}_1\ \ \textit{variable}_1)\ \{ \\
\quad \textit{remedyBody}_1 \\
\}\ \textbf{catch}\ (\textit{exceptionType}_2\ \ \textit{variable}_2)\ \{ \\
\quad \textit{remedyBody}_2 \\
\}\ \ldots \\
\quad \ldots
\end{array}
$$

Throwing Exceptions

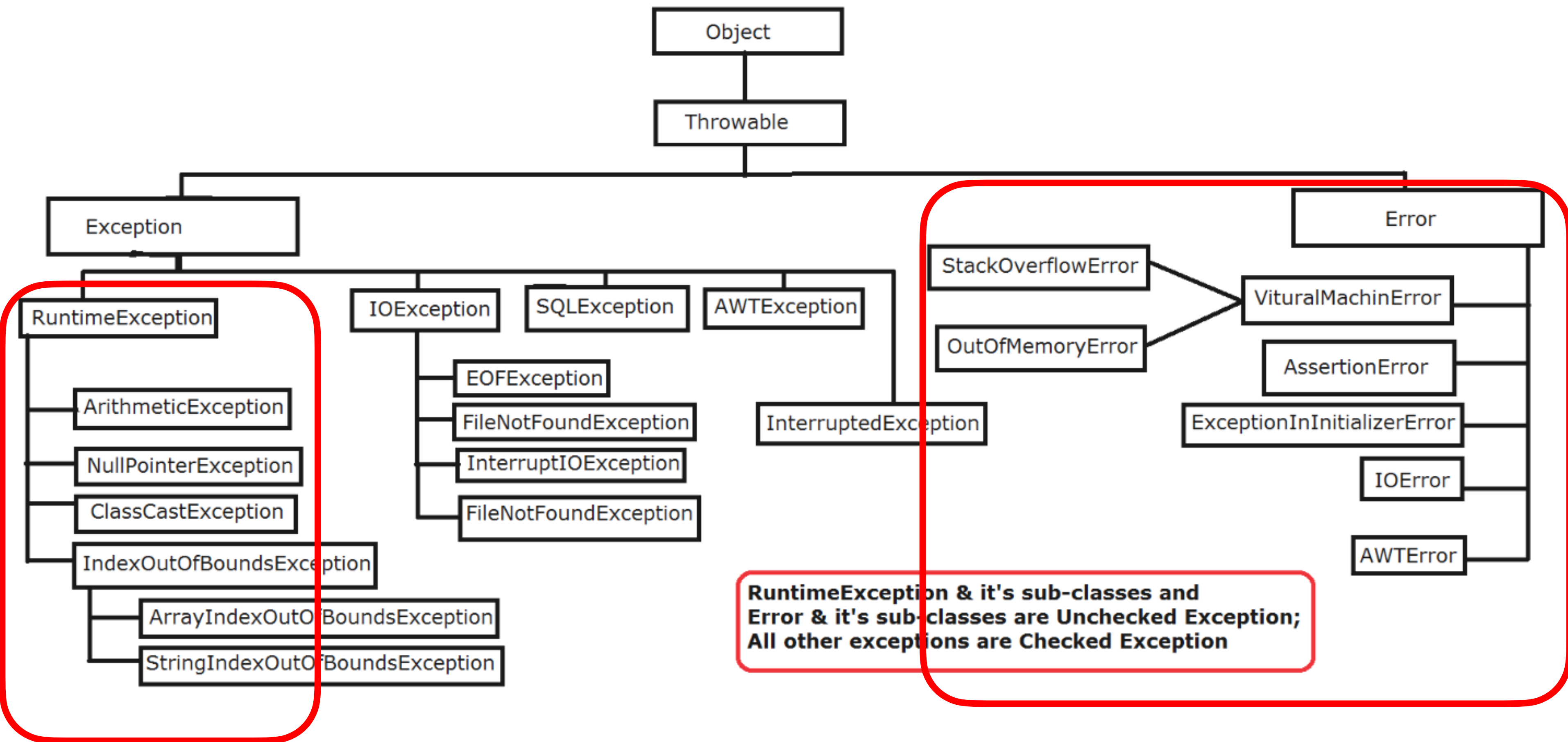
- An exception is thrown
  - implicitly by the JVM because of errors
  - explicitly by code
- If your code throws an exception it **<u>must</u>** catch that exception somewhere else
- Method signature – `throws`

```
public static int parseInt(String s)
throws NumberFormatException
```

Exceptions Example

```
public class ExceptThrower {
    public int divv(int numer, int denom) {
        try {
            return numer / denom;
        } catch (ArithmeticException e) {
            System.err.println("Caught in Func " + e);
        }
        return 0;
    }
    public int divvTh(int numer, int denom) throws ArithmeticException{
        return numer / denom;
    }
    public static void main(String[] args) {
        ExceptThrower except = new ExceptThrower();
        except.divv(2, 0);
        try {
            except.divvTh(4,0);
        } catch (ArithmeticException ae) {
            System.err.println("Caught in Main " + ae);
        }}}
```

Java’s Exception Hierarchy



Reading the Keyboard

- `System.in` is, by default, set to receive keyboard input
- Use Scanner to read from keyboard
  - Do NOT use scanner otherwise

```
import java.util.InputMismatchException;
import java.util.Scanner;
public class CatKeyboard {
    public static void main(String[] args) {
        try (Scanner scanner = new Scanner(System.in);) {
            System.out.print("Enter name: ");
            String name = scanner.next();
            System.out.print("Enter Id:");
            int iid = scanner.nextInt();
            System.out.print("Enter breed: ");
            String breed = scanner.next();
            Cat ccat = new Cat(name, (""+iid), breed);
            System.out.println(ccat);
        } catch (InputMismatchException ime) {
            System.out.println(ime);
        } finally {}}}
```

Handling Exceptions — alternate cat try-catch

```
public Cat() throws InputMismatchException {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter name: ");
    name = scanner.next();
    System.out.print("Enter Id:");
    iD = "" + scanner.nextInt();
    System.out.print("Enter breed: ");
    breed = scanner.next();
}

public Cat getCat() {
    try {
        Cat c = new Cat();
        return c;
    } catch (InputMismatchException ime) {
        System.err.println(ime);
    }
    return null;
}
```

Exceptions should be handled as soon as possible.

try-catch should enclose as little code as possible

Reading from Files

```
public void readOneLineTC()
 {
     BufferedReader br;
     try {
         br = new BufferedReader(
                new FileReader(fileName));
         br.readLine();
     } catch (FileNotFoundException fnf) {
         System.err.println("No file " + e);
     }  catch (IOException e) {
         System.err.println("Reading " + e);
     } finally {
         if (br!=null) {
             try {
                 br.close();
             } catch (IOException ioe) {
                 System.err.println("Close" + ioe);
             }
         }
     }
 }
```

```
public void readOneLineTCR(
    {
        try (BufferedReader br = new BufferedReader(
                    new FileReader(fileName));) {
            br.readLine();
            // close unnecessary in this formulation
        } catch (FileNotFoundException e) {
            System.err.println("Open " + e);
        } catch (IOException e) {
            System.err.println("Reading " + e);
        }
    }
```

finally == code that WILL be executed. Optional part of try-catch

Close can throw an exception so it too must be caught. Sigh

if time, write program to demo try/catch/fianlly

Software Design Goals

- Robustness
  - software capable of error handling and recovery
  - programs should never crash
    - ending abruptly is not crashing
- Adaptability
  - software able to evolve over time and changing conditions (without huge rewrites)
- Reusability
  - same code is usable as component of different systems in various applications
  - The story of Mel — https://www.cs.utah.edu/~elb/folklore/mel.html

OOP Design Principles

- Modularity
  - programs should be composed of “modules” each of which do their own thing
    - each module is separately testable
  - Large programs are built by assembling modules
  - Objects (Classes) are modules
- Abstraction
  - Get to the core — non-removable essence of a thing
  - Most pencils are yellow, but yellowness does not required
- Encapsulation
  - Nothing outside a class should know about how the class works.
    - For instance, does the Object class have any instance variables. (Of what type?)
  - Allows programmer to totally change internals without external effect

OOP Design

- Responsibilities/Independence: divide the work into different classes, each with a different responsibility and are as independent as possible
- Behaviors: define the behaviors for each class carefully and precisely, so that the consequences of each action performed by a class will be well understood by other classes that interact with it.

Class Definition

- Primary means for abstraction in OOP
- Class determines
  - the way state information is stored – via instance variables
  - a set of behaviors – via methods
- Classes encapsulate
  - `private` instance variables
  - `public` accessor methods (getters)

Constructors

- Constructors are never inherited
- A class may invoke the constructor of the class it extends via a call to `super` with the appropriate parameters
  - e.g. `super()`
  - `super` must be in the first line of constructor
  - If no explicit call to `super`, then an implicit call to the zero-parameter `super` will be made
- A class make invoke other constructors of their own class using `this()`
  - `this` must be first
  - Cannot explicitly use both `super` and `this` **in single constructor**
  - See FileOpen.java for example

Lab

Do Feb 16 lab from the class website.

You will use most of this in HW 1 which just adds comments and submits using the homework submission system.