



OOP Design Principles II

EECS 3311

Ilir Dema

demailir@eecs.yorku.ca

Outlines

- **Overview of Java**
 - Java Characteristics
 - Process of Run Java Code
 - Java Class and API
 - Primitive Types
 - Java Operators
 - Access modifiers
 - Java Utils
- **OOP Design Principals**
 - Polymorphism
 - Inheritance
 - Abstraction
 - Generics
 - Exception
- **Junit**
 - Strategies to write good Junit test cases

Acknowledgement

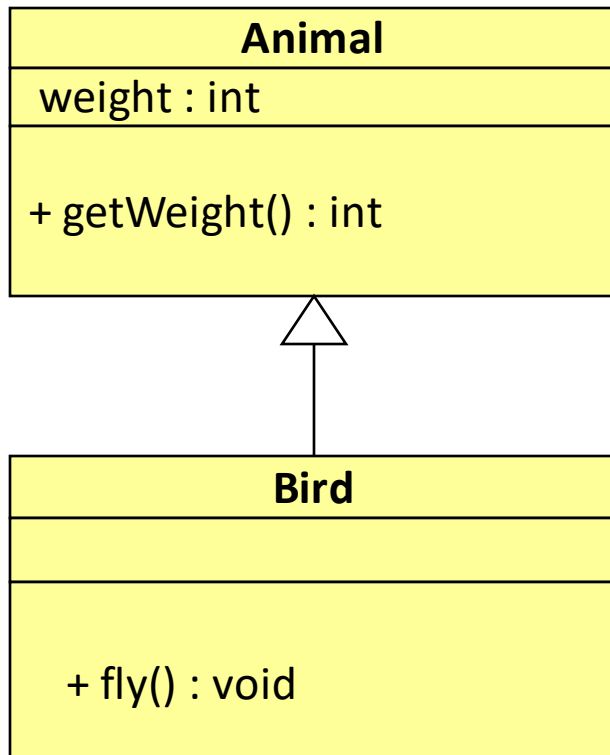
- Some materials/examples used in this lecture come from ICSH22@UCI, ECE444@UT, CS446@UW, CSE331@UW

Polymorphism

- Polymorphism: *many* (poly) *shapes* (morph)
- Enables you to “program in the general” rather than “program in the specific.”
 - Based in Dependency Inversion Principle
- There are two kinds of polymorphism:
 - Overriding (via inheritance)
 - Replacing an inherited method with another having the same signature
 - Overloading
 - Two or more methods with different signatures
- Can significantly simplify programming.

Inheritance

- Inheritance relationships are often shown graphically in a *class diagram*, with the arrow pointing to the parent class



**Inheritance should create
an *is-a* relationship,
meaning the child *is a*
more specific version of
the parent**

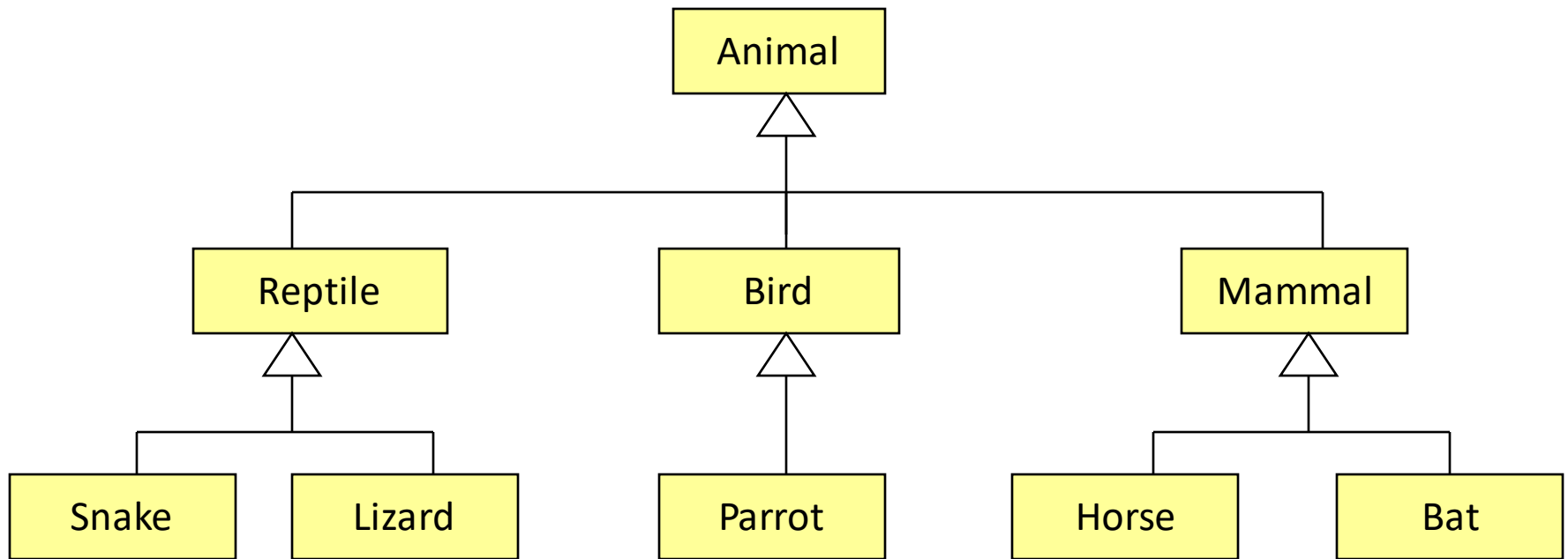
Deriving Subclasses

- In Java, we use the reserved word *extends* to establish an inheritance relationship

```
class Animal {  
    int weight;  
  
    public int getWeight() {  
        return this.weight;  
    }  
}  
  
class Bird extends Animal {  
    public void fly() {  
        // ...  
    }  
}
```

Class Hierarchy

- A child class of one parent can be the parent of another child, forming *class hierarchies*



Class Hierarchy

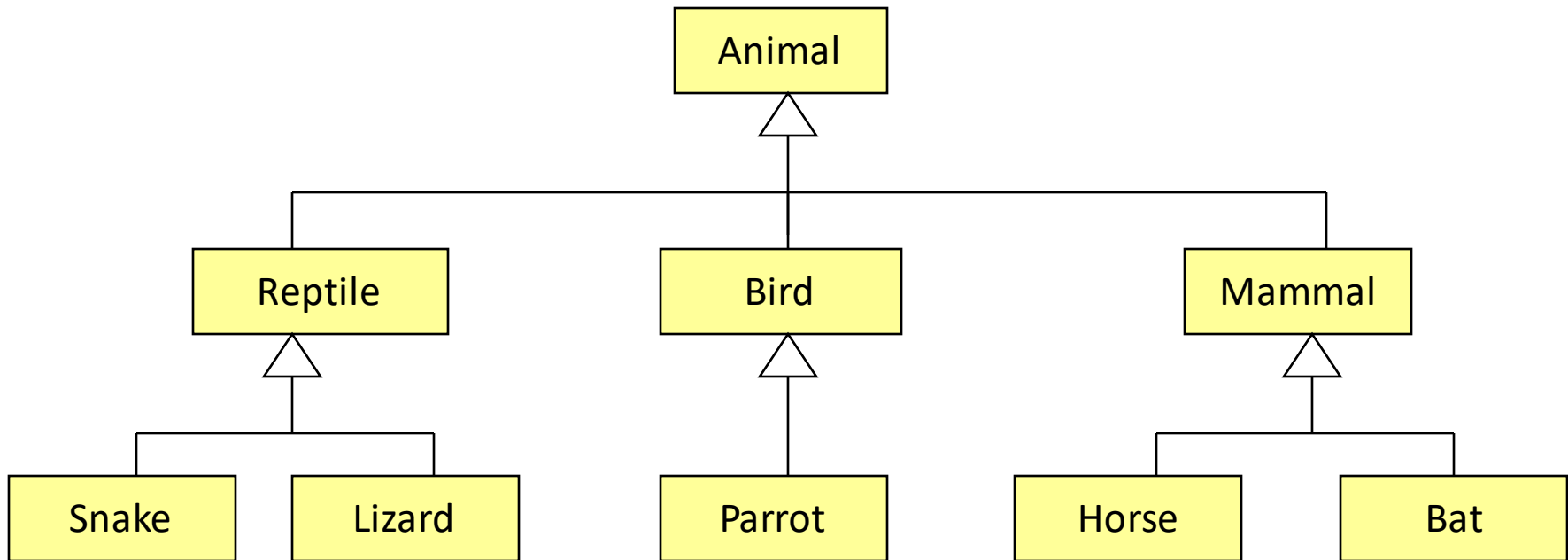
- Good class design puts all common features as high in the hierarchy as reasonable
- **Inheritance is transitive**
 - An instance of class Parrot is also an instance of Bird, an instance of Animal, ..., and an instance of class Object

Defining Methods in the Child Class: Overriding by Replacement

- A child class can *override* the definition of an inherited method in favor of its own
 - that is, *a child can redefine a method that it inherits from its parent*
 - the new method must have the *same signature* as the parent's method, but can have different code in the body
- In java, all methods except of constructors override the methods of their ancestor class by **replacement**. E.g.:
 - the Animal class has method eat()
 - the Bird class has method eat() and Bird extends Animal
 - variable *b* is of class Bird, i.e. Bird b = ...
 - b.eat() simply invokes the eat() method of the Bird class
- *If a method is declared with the final modifier, it cannot be overridden*

Recap: Class Hierarchy

- In Java, a **class can extend a single other class**
(If none is stated then it implicitly extends an Object class)



- Imagine what would happen to method handling rules if every class could extend two others...
(Answer: It would create problems!)

Overloading vs. Overriding

- **Overloading** deals with multiple methods in the same class with the same name but different signatures
- **Overloading** lets you define a similar operation in different ways for different data

- **Overriding** deals with two methods, one in a parent class and one in a child class, that have the same signature
- **Overriding** lets you define a similar operation in different ways for different object types

Controlling Inheritance

- Visibility modifiers determine which class members are accessible and which do not
- Members (variables and methods) declared with public visibility are accessible, and those with private visibility are not
- Problem: How to make class/instance variables visible only to its subclasses?
- Solution: Java provides a third visibility modifier that helps in inheritance situations: **protected**

Multiple Inheritance

- “**diamond problem**”, where it may be **ambiguous** as to which parent class a particular feature is inherited from if more than one parent class implements the same feature.
- **Java doesn't support Multiple Inheritance**, i.e., can not extends more than one superclass
- **C++, Perl, Python support multiple inheritance**

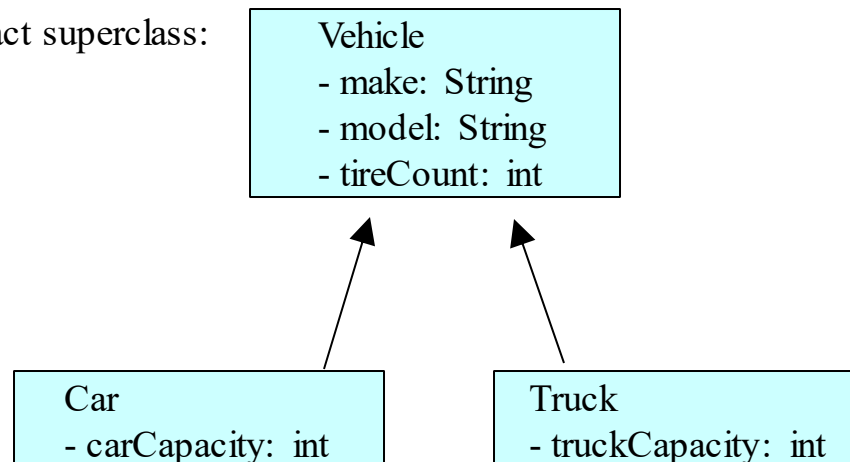
What is an Abstract class?

- Superclasses are created through the process called “**generalization**”
 - **Common features** (methods or variables) are factored out of object classifications (ie. classes).
 - Those features are formalized in a class, i.e., *superclass*
 - The classes from which the common features were taken become *subclasses* to the newly created super class
- Superclass does not have a "meaning" or does not directly relate to a "thing" in the real world
 - It is an artifact of the generalization process
- Because of this, abstract classes *cannot be instantiated*
 - They act as place holders for abstraction

Abstract Class Example

- In the following example, the subclasses represent objects taken from the problem domain.
- The superclass represents an abstract concept that does not exist "as is" in the real world.

Abstract superclass:



What Are Abstract Classes Used For?

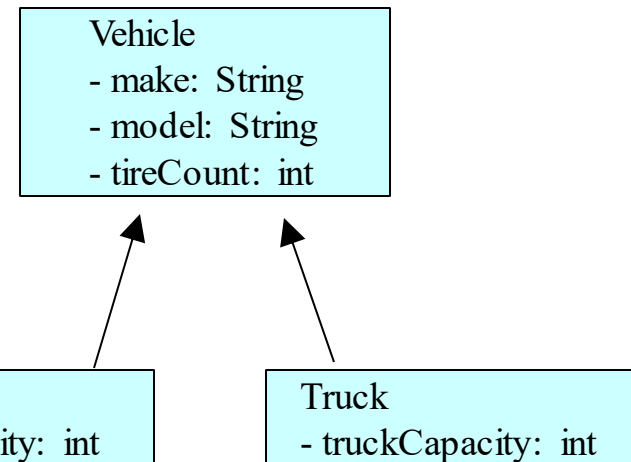
- Abstract classes are used heavily in Design Patterns
 - **Creational Patterns:** Abstract class provides interface for creating objects. The subclasses do the actual object creation.
 - **Structural Patterns:** How objects are structured is handled by an abstract class. What the objects do is handled by the subclasses.
 - **Behavioural Patterns:** Behavioural interface is declared in an abstract superclass. Implementation of the interface is provided by subclasses.
- **In general:** When one needs to generalize (abstract, factor out) common properties and/or behaviours of conceptually related classes
- Be careful not to overuse abstract classes
 - Every **abstract class increases** the complexity of your design
 - Every **subclass increases** the complexity of your design
 - Ensure that you receive acceptable return in terms of functionality given the added complexity.

Defining Abstract Classes

```
public abstract class Vehicle
{
    private String make;
    private String model;
    private int tireCount;
    [...]
```

```
public class Car extends Vehicle
{
    private int carCapacity;
    [...]
```

```
public class Truck extends Vehicle
{
    private int truckCapacity;
    [...]
```



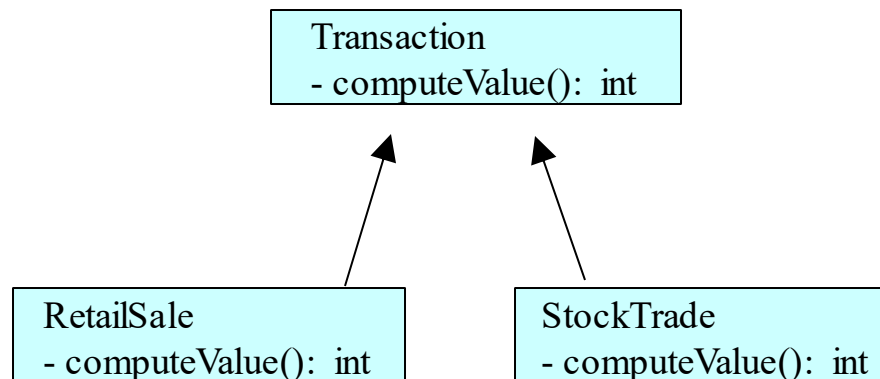
Often referred to as "concrete" classes

Abstract Methods

- Methods can also be abstracted
 - An abstract method is one to which a signature has been provided, but *no implementation* for that method is given.
 - An Abstract method is a placeholder. It means that we *declare that a method must exist*, but there is no meaningful implementation for that methods within this class
- **Any class which contains an abstract method MUST also be abstract**
 - Any class which has an incomplete method definition cannot be instantiated (i.e., it is abstract)
- **Abstract classes can contain both concrete and abstract methods.**
 - If a method can be implemented within an abstract class, and implementation should be provided.

Abstract Method Example

- In the following example, a Transaction's value can be computed, but there is no meaningful implementation that can be defined within the Transaction class.
 - How a transaction is computed is dependent on the transaction's type



Defining Abstract Methods

```
public abstract class Transaction
{
    public abstract int computeValue();
}
```

Note: no implementation

```
public class RetailSale extends Transaction
{
    public int computeValue()
    {
        [...]
    }
}
```

```
public class StockTrade extends Transaction
{
    public int computeValue()
    {
        [...]
    }
}
```

Transaction
- computeValue(): int

RetailSale
- computeValue(): int

StockTrade
- computeValue(): int



What is an Interface?

- An interface is similar to an abstract class with the following exceptions:
 - *All methods defined in an interface are abstract.* Interfaces can contain no implementation.
 - *Interfaces cannot contain instance variables.* However, *they can contain public static final variables* (ie. constant class variables)
 - Note: this is changing in the latest versions of Java. In this course, we maintain the traditional definition of the interface.
- Interfaces are declared using the "*interface*" keyword
 - If an interface is public, it must be contained in a file which has the same name.
- Interfaces are more abstract than abstract classes. There is no hierarchy among them. Interfaces can extend other interfaces. This *creates* a parent-child hierarchy.
- Interfaces are implemented by classes using the "*implements*" keyword.

Implementing Interfaces

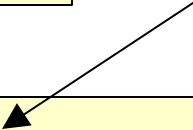
- *A Class can only **extends** from one superclass.* However, a class may **implement** several Interfaces
 - The interfaces that a class implements are **separated by commas**
- Any class which implements an interface must provide an **implementation for all methods defined within the interface.**
 - NOTE: if an abstract class implements an interface, it NEED NOT implement all methods defined in the interface. HOWEVER, each concrete subclass MUST implement the methods defined in the interface.
- An interface can extend multiple interfaces
 - Interfaces can inherit method signatures from other interfaces.

Declaring an Interface

In Steerable.java:

```
public interface Steerable
{
    public void turnLeft(int degrees);
    public void turnRight(int degrees);
}
```

When a class "implements" an interface, the compiler ensures that it provides an implementation for all methods defined within the interface.



In Car.java:

```
public class Car extends Vehicle implements Steerable
{
    public int turnLeft(int degrees)
    {
        [...]
    }

    public int turnRight(int degrees)
    {
        [...]
    }
}
```

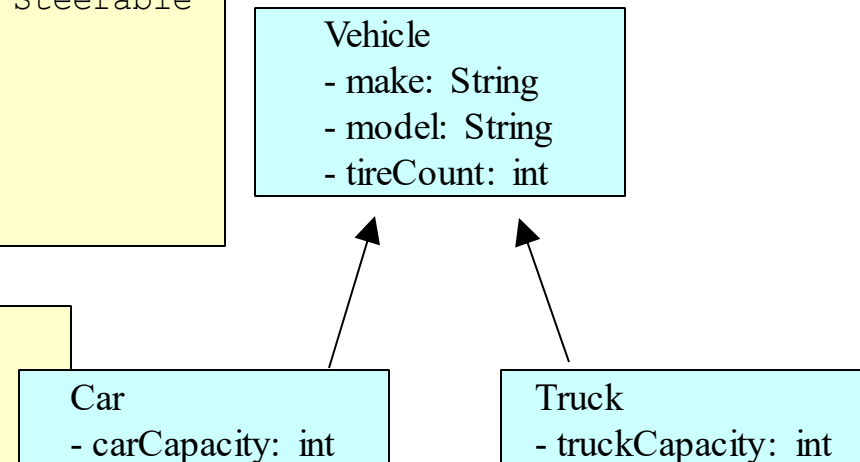
implement Interfaces and extends Abstract Class

- If a superclass implements an interface, it's subclasses also implement the interface

```
public abstract class Vehicle implements Steerable  
{  
    private String make;  
    [...]  
}
```

```
public class Car extends Vehicle  
{  
    private int carCapacity;  
    [...]  
}
```

```
public class Truck extends Vehicle  
{  
    private int truckCapacity;  
    [...]  
}
```



Abstract Classes Versus Interfaces

- When should one use an Abstract class instead of an interface?
 - If the subclass-superclass relationship is genuinely an "is a" relationship.
 - If the abstract class can provide an implementation at the appropriate level of abstraction
- When should one use an interface in place of an Abstract Class?
 - When the level of abstraction of the methods is such that they can be used by a huge variety of other abstractions
 - When the subclass needs to inherit from another class
 - When you cannot reasonably implement any of the methods

Why generic programming

Background

- old version 1.4 Java collections were *Object*-based and required the use of ugly casts
 - cannot specify the exact type of elements
 - must cast to specific classes when accessing

Java generics

- lets you write code that is safer and easier to read
- is especially useful for general data structures, such as ArrayList
- generic programming = programming with classes and methods parameterized with types

Generics: Parametric polymorphism

- **parametric polymorphism:** Ability for a function or type to be written in such a way that it handles values identically **without depending on knowledge of their types**.
 - Such a function or type is called a *generic function* or **data type**.
 - first introduced in 1976
 - now part of many other languages (Haskell, Java, C++ ...)
- **Motivation:** Parametric polymorphism allows you to write flexible, general code without sacrificing type safety.
 - Most commonly used in Java with collections.

Java collections ≤ v1.4

- The initial Java collections stored values of type **Object**.
 - They could store any type, since all types are subclasses of Object.
 - But you had to cast the results, which was tedious and error-prone.
 - Examining the elements of a collection was not type-safe.

// in Java 1.4:

```
ArrayList names = new ArrayList(); <=> new ArrayList<Object>();  
names.add("Marty Stepp");  
names.add("Stuart Reges");  
String teacher = (String) names.get(0);
```

// this will compile but crash at runtime; bad
Point oops = (Point) names.get(1);

Type Parameters (Generics)

```
List<Type> name = new ArrayList<Type>();
```

- Since Java 1.5, when constructing a `java.util.ArrayList`, we specify the type of elements it will contain between `<` and `>`.
 - We say that the `ArrayList` class accepts a **type parameter**, or that it is a **generic** class.
 - Use of the "raw type" `ArrayList` without `<>` leads to warnings.

```
List<String> names = new ArrayList<String>();  
names.add("Marty Stepp");  
names.add("Stuart Reges");  
String teacher = names.get(0);           // no cast  
Date oops = (Date) names.get(1);        // error
```

Implementing generics

// a parameterized (generic) class

```
public class name<Type> {
```

or

```
public class name<Type, Type, ..., Type> {
```

- By putting the **Type** in `< >`, you are demanding that any client that constructs your object must supply a type parameter.
 - You can require multiple type parameters separated by commas.
- The rest of your class's code can refer to that type by name.
 - The convention is to use a 1-letter name such as:
T for Type, E for Element, N for Number, K for Key, or V for Value.
- The type parameter is *instantiated* by the client. (e.g. E → String)

Generics and arrays

```
public class Foo<T> {  
    private T myField;           // ok  
    private T[] myArray;        // ok  
  
    public Foo(T param) {  
        myField = new T();      // error  
        myArray = new T[10];    // error  
    }  
}
```

- You cannot create objects or arrays of a parameterized type.

Generics/arrays, fixed

```
public class Foo<T> {  
    private T myField;           // ok  
    private T[] myArray;        // ok  
  
    @SuppressWarnings("unchecked")  
    public Foo(T param) {  
        myField = param;        // ok  
        T[] a2 = (T[]) (new Object[10]); // ok  
    }  
}
```

- But you can create variables of that type, accept them as parameters, return them, or create arrays by casting `Object[]`.

Comparing generic objects

```
public class ArrayList<E> {  
    ...  
    public int indexOf(E value) {  
        for (int i = 0; i < size; i++) {  
            // if (elementData[i] == value) {  
                if (elementData[i].equals(value)) {  
                    return i;  
                }  
            }  
        }  
        return -1;  
    }  
}
```

- When testing objects of type E for equality, must use `equals`

A generic interface

// Represents a list of values.

```
public interface List<E> {  
    public void add(E value);  
    public void add(int index, E value);  
    public E get(int index);  
    public int indexOf(E value);  
    public boolean isEmpty();  
    public void remove(int index);  
    public void set(int index, E value);  
    public int size();  
}
```

```
public class ArrayList<E> implements List<E> { ...
```

```
public class LinkedList<E> implements List<E> { ...
```

Generic methods

```
public static <Type> returnType name (params)
{
```

- When you want to make just a single (often static) method generic in a class, precede its return type by type parameter(s).

```
public class Collections {
    ...
    public static <T> void copy(List<T> dst, List<T> src) {
        for (T t : src) {
            dst.add(t);
        }
    }
}
```

Bounded type parameters

<Type extends SuperType>

- An upper bound; accepts the given supertype or any of its subtypes.

<Type super SuperType>

- A lower bound; accepts the given supertype or any of its supertypes.

- Example:

```
public class TreeSet<T extends E> {  
    ...  
}
```

Complex bounded types

- `public static <T> void copy(
 List<T2 super T> dst, List<T3 extends T> src)`
- Copy all elements from src to dst. For this to be reasonable, dst must be able to safely store anything that could be in src. This means that all elements of src must be of dst's element type or a subtype.

Wildcards

- ? indicates a *wild-card* type parameter, one that can be any type.
 - `List<?> list = new List<?>(); // anything`
- Difference between `List<?>` and `List<Object>` :
 - ? can become any particular type; `Object` is just one such type.
 - `List<Object>` is restrictive; wouldn't take a `List<String>`
- Difference between. `List<Foo>` and `List<? extends Foo>`:
 - The latter binds to a particular `Foo` subtype and allows ONLY that.
 - e.g. `List<? extends Animal>` might store only Giraffes but not Zebras
 - The former allows anything that is a subtype of `Foo` in the same list.
 - e.g. `List<Animal>` could store both Giraffes and Zebras

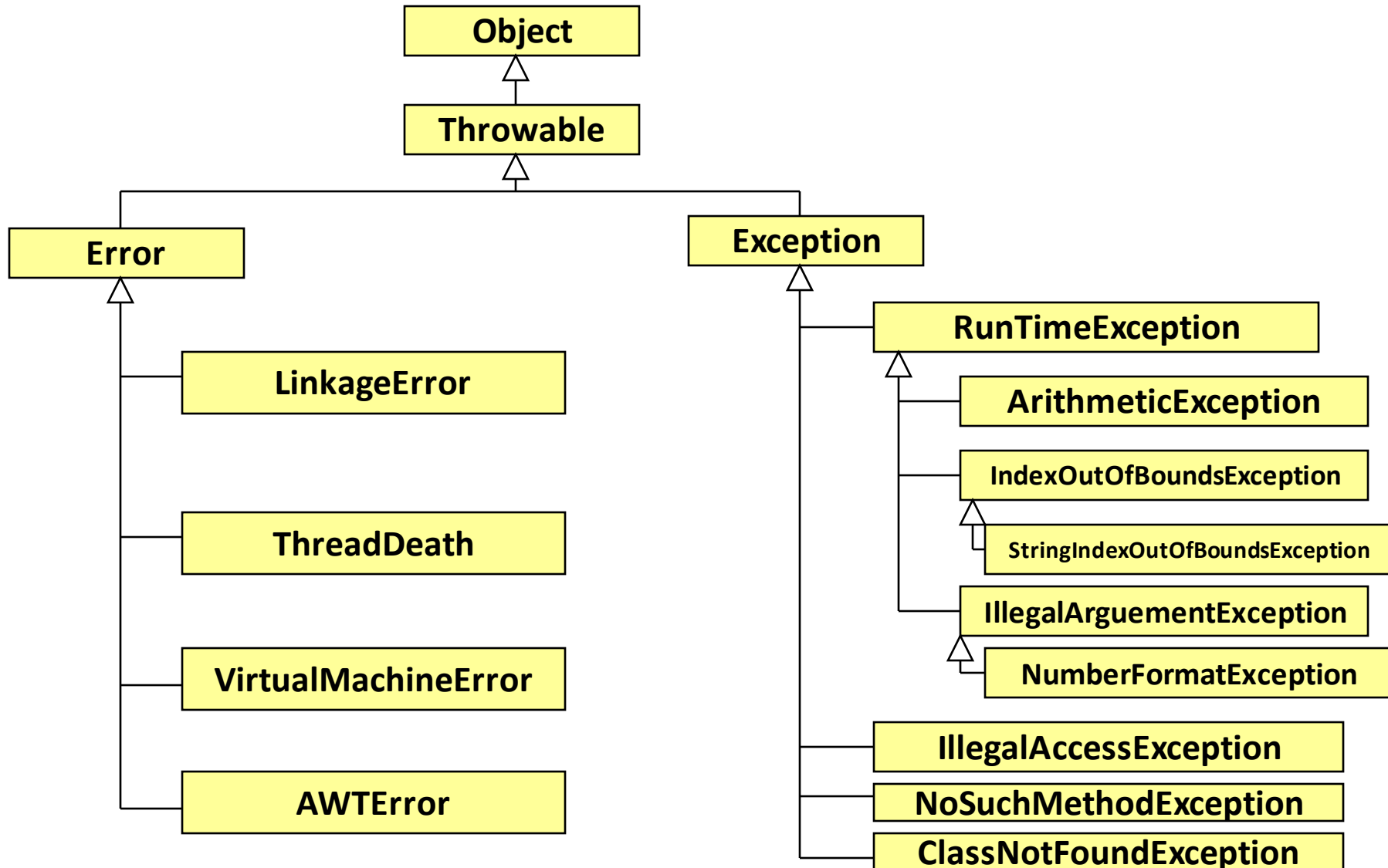
Exceptions

- An *exception* is an *object* that enables a program to handle unusual/erroneous situations

- A method can *throw* an exception, e.g.

```
public void doubleArray( int[] A) throws Exception
{
...
if (Index >= A.length)
    throw new Exception( "array too small " + Index)
...
}
```

Exception is a class, and all Exception-like objects should be subclasses of Exception



Extending Exception Classes

- A method can throw several exceptions (each of which is a subclass of the `Exception` class), e.g.:

```
public void scaleArray( int[] A, int s) throws ArrayRangeException,  
                                              IllegalArgumentException  
{ ...  
  if (Index >= A.length)  
    throw new ArrayRangeException( "array too small " + Index)  
  ... }
```
- Class ***ArrayRangeException***, ***IllegalArgumentException*** must be a subclass of (predefined) class *Exception*, and one of its constructors takes a string argument

Exception Propagation

A (caller) method can deal with an exception thrown by the method it calls in 2 ways:

- A caller method can **ignore** exception handling
 - In this case the exception thrown by the called method will be “passed up” and (effectively) thrown by the caller method
 - This exception propagation will continue until the *main* method which was an access point of the java code, which will throw an error to the user (and print its description in the console output)
- *Except* if any of the methods along the caller/callee chain explicitly **handles** this exception. This breaks the chain of exception propagation, and after the exception is handled, the control returns to normal execution (see next slide).

Exception Handling:

The try and catch Statement

- To handle an exception when it occurs, the line that throws the exception is executed within a *try block*
- A try block must be followed by one or more *catch* clauses, which contain code that processes an exception
- Each catch clause has an associated exception class, e.g. `ArrayRangeException`, `IllegalArgumentException`
- When an exception occurs, processing continues at the first catch clause that *matches* the (most specific) exception class of the exception object which is thrown by any statement in the try block.

Exception Handling:

The **try** , **catch** Statement + **finally**

```
try {  
    int a = b/c;  
    String name = names[a];  
}  
catch (ArithmeticException ax){  
    System.out.println("Division of "+b+" by "+c+" cannot be carried  
    out.");  
}  
catch (ArrayIndexOutOfBoundsException aioobx)  
{  
    System.out.println("The index "+a+" is out of bounds");  
}  
finally { <statement> }
```

The **finally** Clause

- A try statement can have an optional clause designated by the reserved keyword **finally**
- If no exception is generated, the statements in the finally clause are executed after the statements in the try block complete
- Also, if an exception is generated, the statements in the finally clause are executed after the statements in the appropriate catch clause complete

Writing your own exception class

- Create a new class whose name should end with `Exception` like `XXXXException`. This is a name convention.
- Make the class `extends` one of the exceptions which are subtypes of the `java.lang.Exception` class.
- Generally, a custom exception class always extends directly from the `Exception` class.
- Create a constructor with a `String` parameter which is the detail message of the exception. In this constructor, simply call the super constructor and pass the message.

Examples

```
public class StudentNotFoundException extends Exception {  
    public StudentNotFoundException(String message) {  
        super(message);  
    }  
}
```

```
public class StudentManager {  
    public Student find(String studentID) throws StudentNotFoundException {  
        if (studentID.equals("123456")) {  
            return new Student();  
        } else {  
            throw new StudentNotFoundException(  
                "Could not find student with ID " + studentID);  
        }  
    }  
}
```

Examples (cont.)

```
public class StudentTest {  
    public static void main(String[] args) {  
        StudentManager manager = new StudentManager();  
  
        try {  
            Student student = manager.find("0000001");  
  
        } catch (StudentNotFoundException ex) {  
            System.err.print(ex);  
        }  
    }  
}
```

Output: Could not find student with ID 0000001