Object-Oriented Programming

Instructor: Andy Mirzaian

Object-Oriented Software Design

- **Responsibilities**:
  Divide the work into different actors, each with a different responsibility. These actors become classes.

- **Independence**:
  Define the work for each class to be as independent from other classes 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.

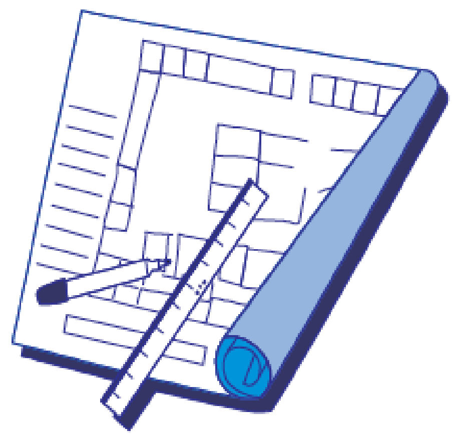
Software must be

- **Correct:** works correctly on all expected inputs.
- **Readable:** easily understandable & verifiable by others.
- **Robust:** capable of handling unexpected inputs that are not explicitly defined for its intended application.
- **Efficient:** makes good use of computing time & memory resources.
- **Adaptable:** able to evolve over time in response to changing conditions in its environment. Is easy to update & debug.
- **Flexible:** easily generalizable to handle many related scenarios.
- **Reusable:** the same code should be usable as a component of different systems in various applications.

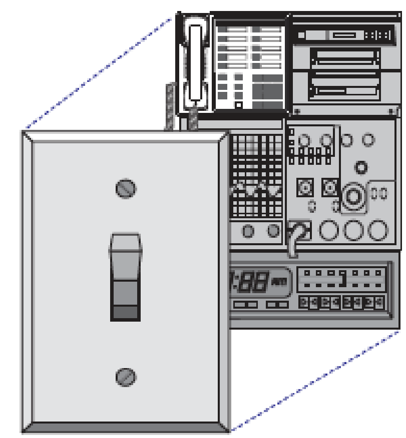
Object-Oriented Design Principles

- Abstraction
- Modularity
- Encapsulation
- Hierarchical Organization

Modularity



Abstraction



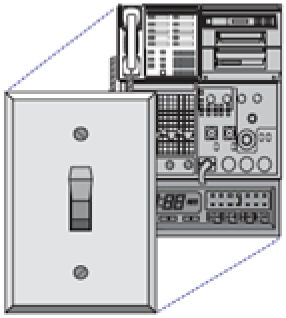
Encapsulation

Abstraction

- **Abstraction** is to distill a system to its most fundamental parts.
  - *The psychological profiling of a programmer is mostly the ability to shift levels of abstraction, from low level to high level. To see something in the small and to see something in the large.*
    – Donald Knuth

Abstraction, 1922.
By *Wassily Kandinsky*

Encapsulation

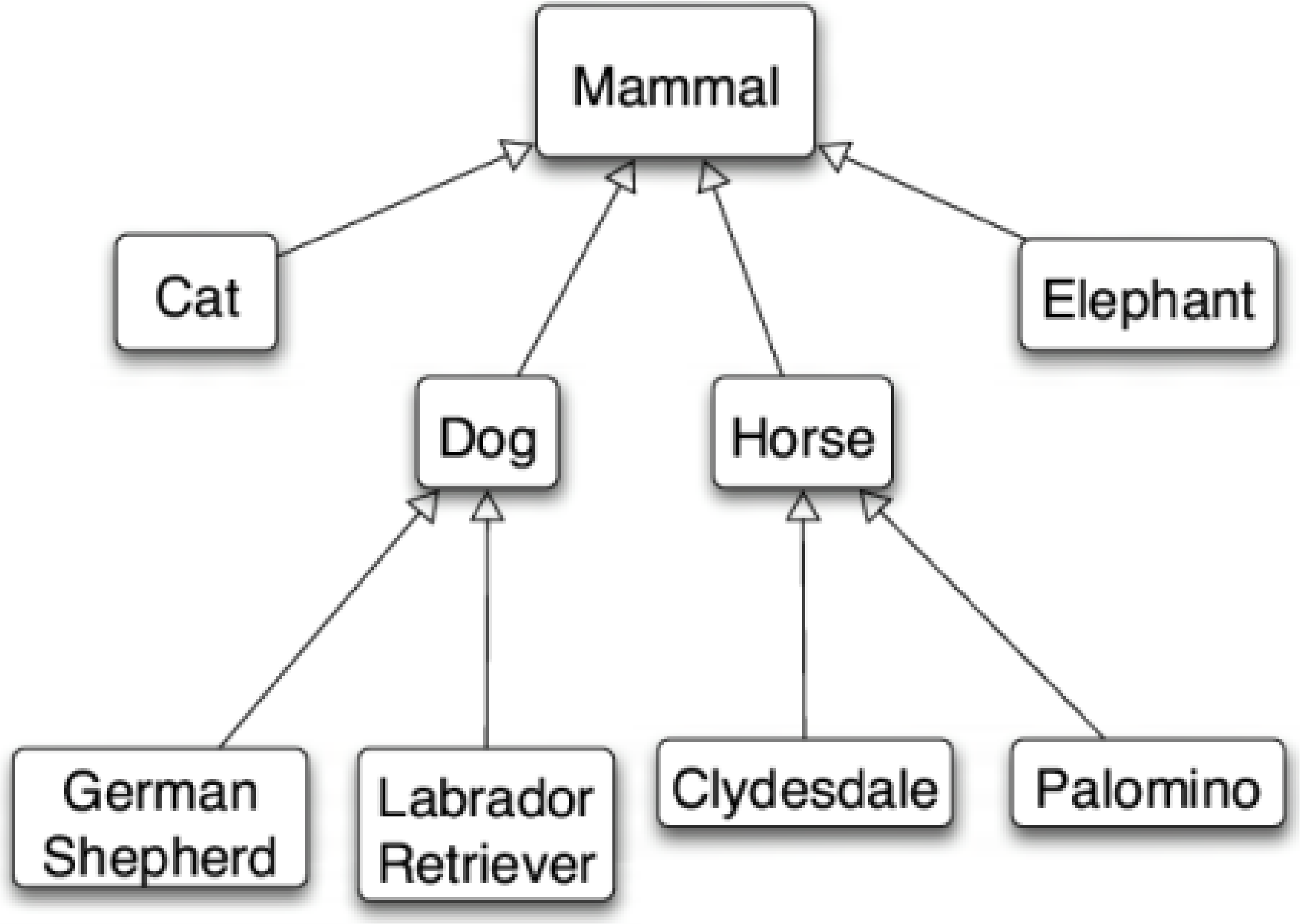


- Information hiding.
- objects reveal only what other objects need to see.
- Internal details are kept private.
- This allows the programmer to implement the object as they wish, as long as the requirements of the abstract interface are satisfied.

Modularity

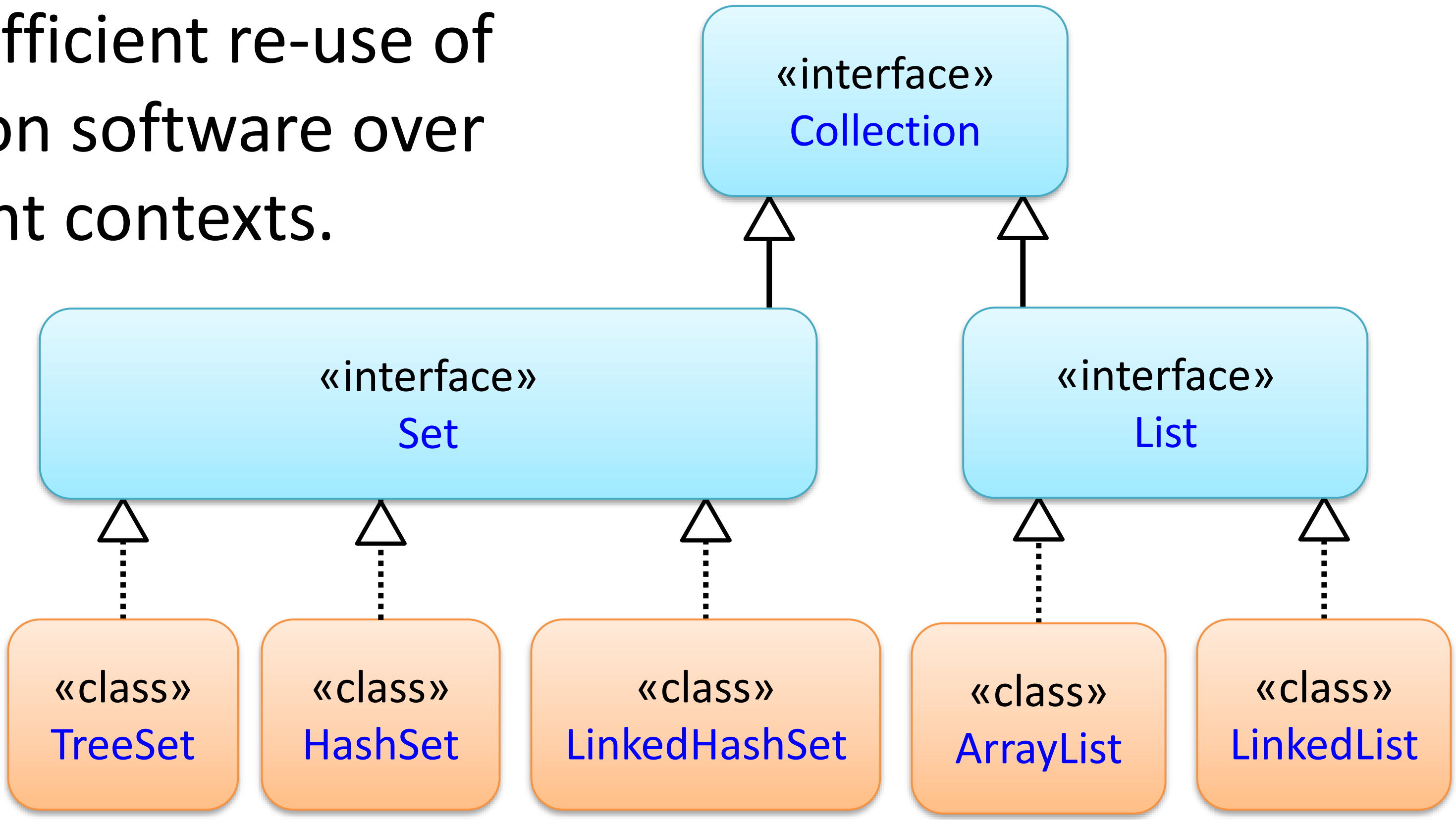
- Complex software systems are hard to conceptualize, design & maintain.
- This is greatly facilitated by breaking the system up into distinct modules.
- Each module has a well-specified role.
- Modules communicate through well-specified interfaces.
- The primary unit for a module in Java is a package.

A Hierarchy

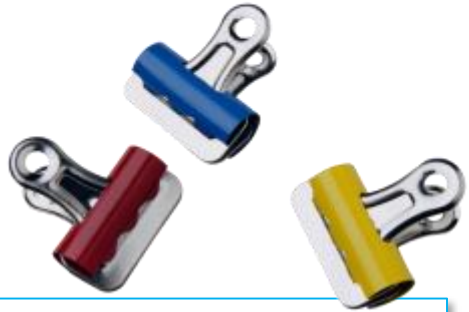

Mammal
Cat
Elephant
Dog
Horse
German Shepherd
Labrador Retriever
Clydesdale
Palomino

Hierarchical Design

Hierarchical class definitions allow efficient re-use of common software over different contexts.



Design Patterns



Algorithmic patterns:

- Recursion
- Amortization
- Divide-and-conquer
- Prune-and-search
- Brute force
- Dynamic programming
- The greedy method

Software design patterns:

- Iterator
- Adapter
- Position
- Composition
- Template method
- Locator
- Factory method

Abstract Data Types

- **Abstraction** is to distill a system to its most fundamental parts.
- Applying the abstraction paradigm to the design of data structures gives rise to **abstract data types** (ADTs) with state (data) & behavior (functionality).
- An ADT is a model of a data structure that specifies the **type** of data stored, the **operations** supported on them, and the types of parameters of the operations.
- An ADT specifies **what** each operation does, but not **how** it does it.
  - The **"how"** is provided by the **software** that implements the ADT.
- The collective set of behaviors supported by an ADT is its **public interface**. The interface guarantees certain **invariants**.
- **Invariant:** a fact about the ADT that is always true,
  e.g., a Date object always represents a valid date.

Class Definitions

- A **class** serves as the primary means for abstraction in OOP.
- In Java, every variable is either a base type or is a reference to an **object** which is an **instance** of some class.
- Each class presents to the outside world a concise and consistent view of the objects that are its instances, without revealing too much unnecessary detail or giving others access to the inner workings of the objects.
- The class definition specifies its members. These are typically **instance variables** (aka, **fields** or **data members**) that any instance object contains, as well as the **methods**, (aka, **member functions**) that the object can execute.

Unified Modeling Language (UML)

A **class diagram** has three parts.

1. The name of the (concrete or abstract) class or interface
2. The recommended instance variables or fields
3. The recommended methods of the class.

class:	CreditCard	
fields:	− customer : String − bank : String − account : String	− limit : int # balance : double
methods:	+ getCustomer( ) : String + getBank( ) : String + charge(price : double) : boolean + makePayment(amount : double)	+ getAccount( ) : String + getLimit( ) : int + getBalance( ) : double

Interfaces

- The main structural element in Java that enforces an application programming interface (API) is an interface.

- An **interface** contains constants & abstract methods with no bodies; all public by default.

- It has no constructors & can't be directly instantiated.

- A class that implements an interface, must implement **all** of the methods declared in the interface (no inheritance); otherwise won't compile.

Abstract Classes

- An **abstract class** also cannot be instantiated, but it can define one or more methods that all implementations of the abstraction will have.
- Their sole purpose is to be extended.
- A class must be a subclass of an abstract class to **extend** it & implement all its abstract methods (or else be abstract itself).

Interfaces & Abstract Classes

- A class that implements an interface, must implement **all** of the methods declared in the interface (no inheritance); otherwise won't compile.
- As a result, unlike abstract classes, interfaces are non-adaptable: you can't add new methods to it without breaking its contract.
- However, interfaces offer great flexibility for its implementers: a class can **implement** any number of interfaces, regardless of where that class is in the class hierarchy.

Inheritance

- is a mechanism for modular and hierarchical organization.
- A (child) **subclass extends** a (parent) **superclass.**
- A subclass inherits (non-constructor) members of its superclass.

- Two ways a subclass can differ from its superclass:
  - Can **extend** the superclass by providing brand-new data members & methods (besides those inherited from the superclass, other than constructors).
  - **Polymorphism:** may specialize an existing behavior by providing a new implementation to **override** an existing non-static method of the superclass .

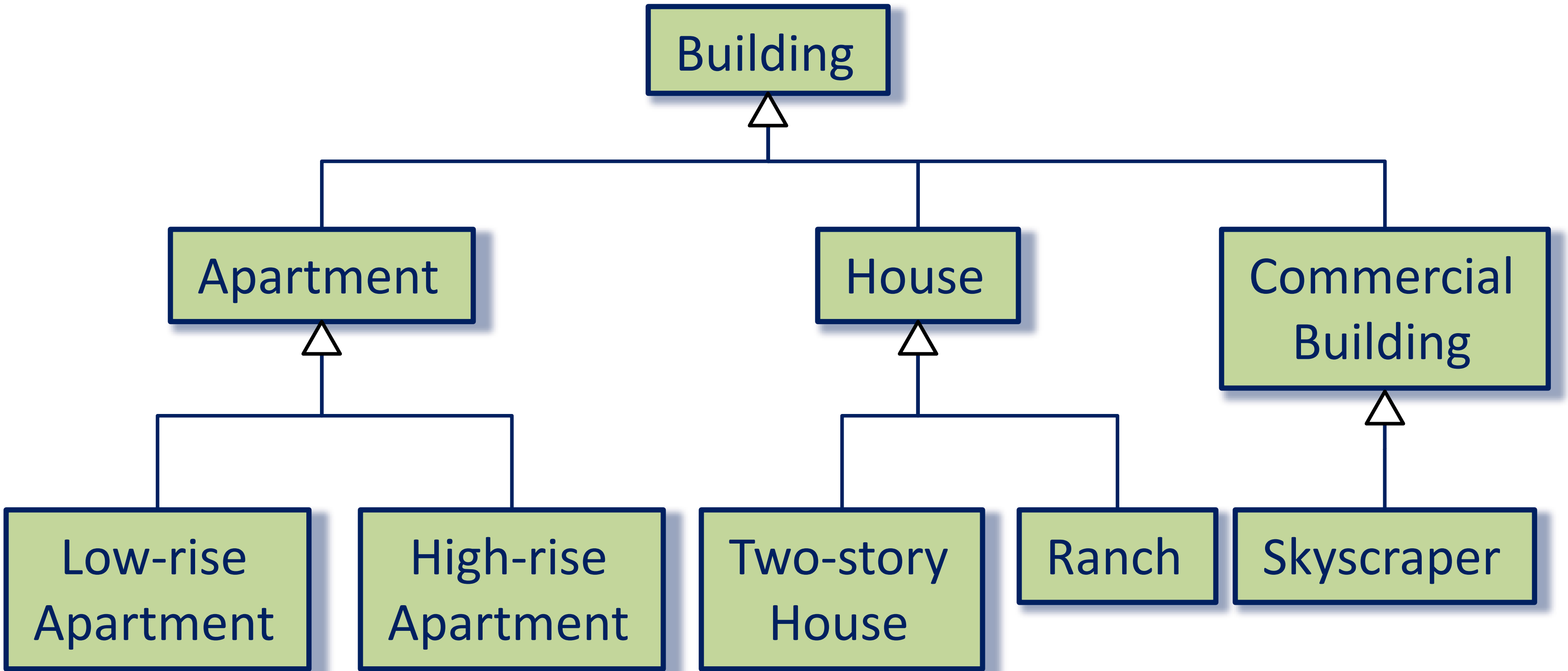
Java is Single Inheritance

- Java (unlike C++) is **single inheritance** OOL:
  any class other than the root class **`Object`**,
  extends exactly one parent superclass.
  That is, Java classes form a **tree hierarchy**.

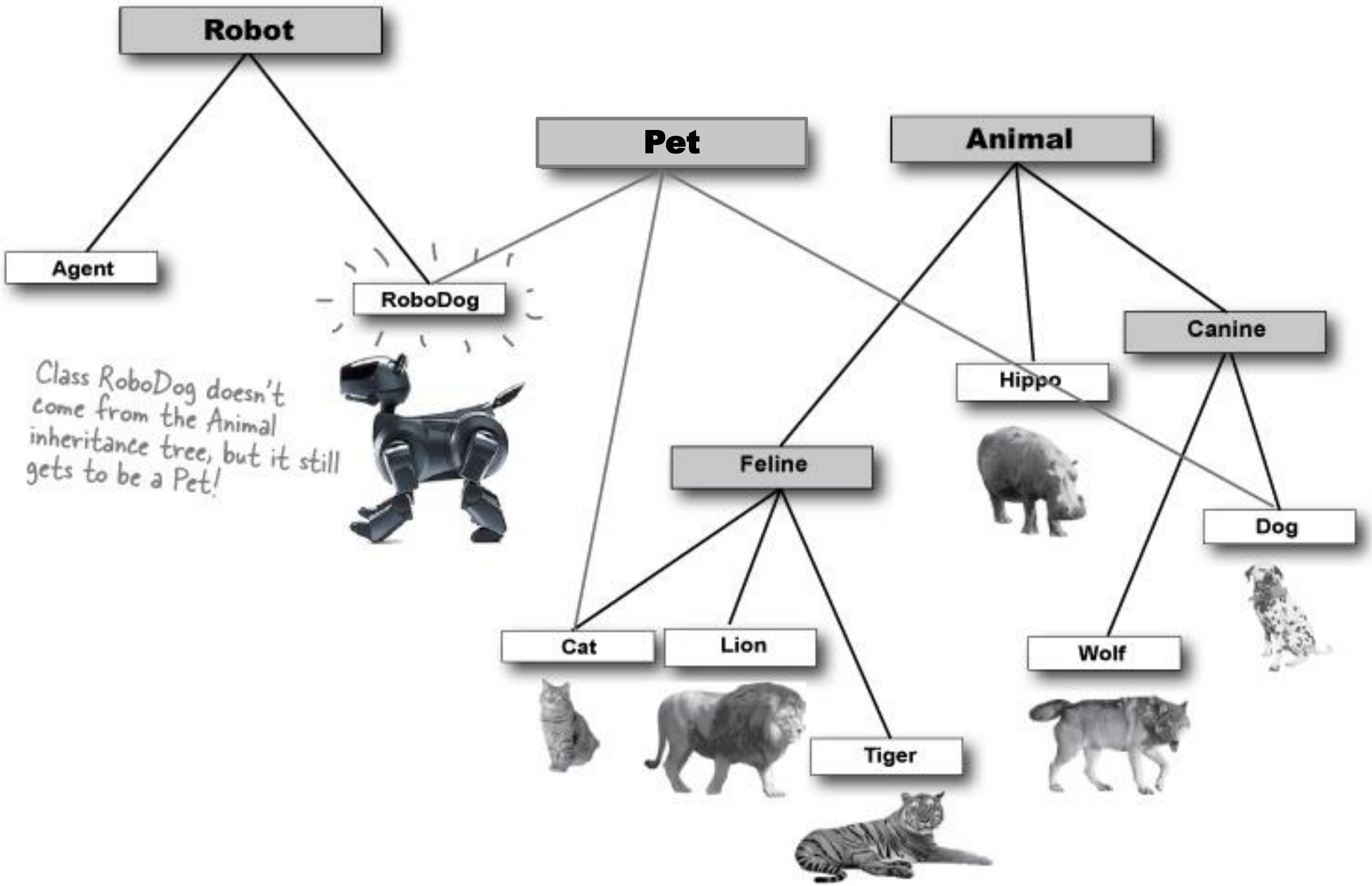
- Regardless of where it is in the inheritance tree,
  a class can **implement several interfaces**.

  This is *multi-role playing* (aka, *mixin*),
  not multiple inheritance.

Class Inheritance Tree Hierarchy



Class/interface DAG Hierarchy



Constructors

- A user can create an instance of a class by using the **new** operator with a method that has the same name as the class.

- Such a method, known as a **constructor**, establishes a new object with appropriate initial values for its instance variables.

Inheritance and Constructors

- **Constructors are never inherited in Java;** hence, every class must define a constructor
  - which can refine a superclass constructor.
  - must properly initialize all class fields, including any inherited fields.
- The first operation within the body of a constructor must be to invoke a constructor of the superclass, which initializes the fields defined in the superclass.
- A constructor of the superclass is invoked explicitly by using the keyword **super** with appropriate parameters.
- If a constructor for a subclass does not make an explicit call to **super** or **this** as its first command, then an implicit call to **super**( ), the zero-parameter version of the superclass constructor, will be made.

Polymorphism

- **Polymorphism:** means taking on many forms.
- **Example:** *Super var =* ***new*** *Sub*(…);

  says *var* is declared as *Super* type, but is instanceof and

  references an object of *Sub* type.

- *var* is **polymorphic**; it can take one of many forms, depending on the specific class or subclass of the object to which it refers at runtime.

Dynamic dispatch

- With polymorphism, one method works on many classes, even if the classes need different implementations of that method.
- **Dynamic dispatch** is a process used by JVM at runtime to call the version of the overriden method most specific to actual (dynamic) type, not declared (static) type, of the polymorphic variable *var*.

- **Example:** *Super var =* ***new*** *Sub(…);*

  Suppose we call *var.myMethod*

  and at runtime *(var instanceof Sub)* is ***true***.

  Will JVM execute *var.(Sub.myMethod)* or *var.(Super.myMethod)* ?

  - JVM calls *Sub.myMethod,* since *var* refers to an instance of *Sub,* even though its static type is Super.

Overriding vs overloading

- **Overriden** method selection is **dynamic** (uses dynamic dispatch)
- **Overloaded** method selection is **static,**
  based on compile-time type of the parameters.
- Because overriding is the norm and overloading is the exception, overriding sets people’s expectations for the behavior of method invocation.
- Most often, instead of overloading, we can use different names.
- **Constructors** can’t use different names & are typically overloaded, but fortunately they cannot be overriden!

Motto: avoid confusing uses of overloading.

- See more examples on the following pages.

Example: Overriding

```
//   ---------------------------- What does this program print?
public class  Wine {
      String  name( ) { return "wine" ; }
}
public class  SparklingWine  extends  Wine {
      @Override  String  name( ) { return "sparkling wine" ; }
}
public class  Champagne  extends  SparklingWine {
      @Override  String  name( ) { return "champagne" ; }
}
public class Overriding {
      public static  void  main(String[ ] args) {
            Wine[ ]  wines = { new Wine(),  new SparklingWine(),  new Champagne() } ;
            for  (Wine wine : wines)    System.out.println( wine.name() ) ;
      }
}
```

output:

```
wine
sparkling wine
champagne
```

Example: Overloading

```
// ------------------------ Broken! – What does this program print?
public class WineRegion {
    public static String region ( Wine w ) { return "Napa Valley" ; }
    public static String region ( SparklingWine s ) { return "Niagara" ; }
    public static String region ( Champagne c ) { return "France" ; }

    public static void main(String[ ] args) {
        Wine[ ] wines = {
            new Wine() ,
            new SparklingWine () ,
            new Champagne ()
        } ;
        for ( Wine w : wines ) System.out.println( region(w) ) ;
    }
}
```

output:

```
Napa Valley
Napa Valley
Napa Valley
```

Example: Overloading - fixed

```
// Fixed by a single method that does an explicit  instanceof  test
public class   WineRegion {
        public static   String   region ( Wine w ) {
                return  ( w  instanceof  Champagne ) ?  "France"  :
                          ( ( w  instanceof  SparklingWine ) ?  "Niagara"  :  "Napa Valley" );

        }
        public static  void  main(String[ ] args) {
                Wine[ ]  wines = {
                        new Wine() ,
                        new SparklingWine () ,
                        new Champagne ()
                } ;
                for  ( Wine w  :  wines )   System.out.println( region(w) ) ;
        }
}
```

output:

```
Napa Valley
Niagara
France
```

Class definition syntax

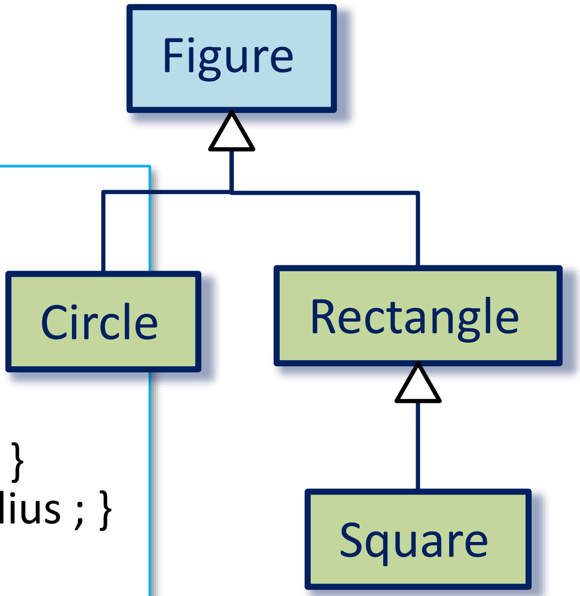
```
class SubClass
      extends SuperClass
      implements Interface1, Interface2, Interface3
      {
            // definitions of non-inherited instance variable
            // subclass constructors
            // overriden superclass methods
            // other, inherited, superclass methods omitted
            // implementation of all interface methods
            // brand-new methods
      }
```

Interface definition syntax

```
interface   YourNewInterface
        extends   YourInterface1 , YourInterface2, YouInterface3
        {
                . . .
        }
```

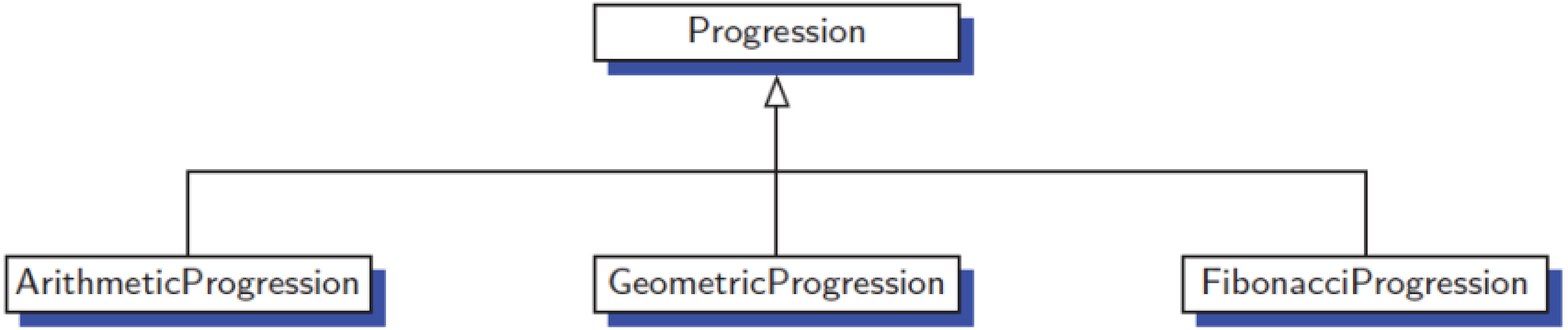
Example

```
abstract class   Figure  {
        abstract double area() ;
}
class Circle extends Figure  {
        final double radius ;
        Circle (double radius)  {  this.radius = radius ; }
        double area()  { return Math.PI * radius * radius ; }
}
class Rectangle extends Figure {
        final double length , width ;
        Rectangle (double length , double width) {
                this.length = length ;
                this.width = width ;
        }
        double area() { return length * width ; }
}
class Square extends Rectangle {
        Square (double side) { super(side , side) ; }
}
```



An Extended Example

- A **numeric progression** is a sequence of numbers, where each number depends on one or more of the previous numbers.
  - An **arithmetic progression** determines the next number by adding a fixed constant to the previous value.
  - A **geometric progression** determines the next number by multiplying the previous value by a fixed constant.
  - A **Fibonacci progression** uses the formula $N_{i+1} = N_i + N_{i-1}$



The Progression Base Class

```
/** Generates a simple progression. By default: 0, 1, 2, ... */
public class Progression {

  // instance variable
  protected long current;

  /** Constructs a progression starting at zero. */
  public Progression( ) { this(0); }

  /** Constructs a progression with given start value. */
  public Progression(long start) { current = start; }

  /** Returns the next value of the progression. */
  public long nextValue( ) {
    long answer = current;
    advance( );    // this protected call is responsible for advancing the current value
    return answer;
  }
```

The Progression Base Class, 2

```

  /** Advances the current value to the next value of the progression. */
  protected void advance( ) {
    current++;
  }

  /** Prints the next n values of the progression, separated by spaces. */
  public void printProgression(int n) {
    System.out.print(nextValue( ));          // print first value without leading space
    for (int j=1; j < n; j++)
      System.out.print(" " + nextValue( ));  // print leading space before others
    System.out.println( );                   // end the line
  }
}
```

ArithmeticProgression Subclass

```
public class ArithmeticProgression extends Progression {

  protected long increment;

  /** Constructs progression 0, 1, 2, ... */
  public ArithmeticProgression( ) { this(1, 0); }      // start at 0 with increment of 1

  /** Constructs progression 0, stepsize, 2*stepsize, ... */
  public ArithmeticProgression(long stepsize) { this(stepsize, 0); }      // start at 0

  /** Constructs arithmetic progression with arbitrary start and increment. */
  public ArithmeticProgression(long stepsize, long start) {
    super(start);
    increment = stepsize;
  }

  /** Adds the arithmetic increment to the current value. */
  protected void advance( ) {
    current += increment;
  }
}
```

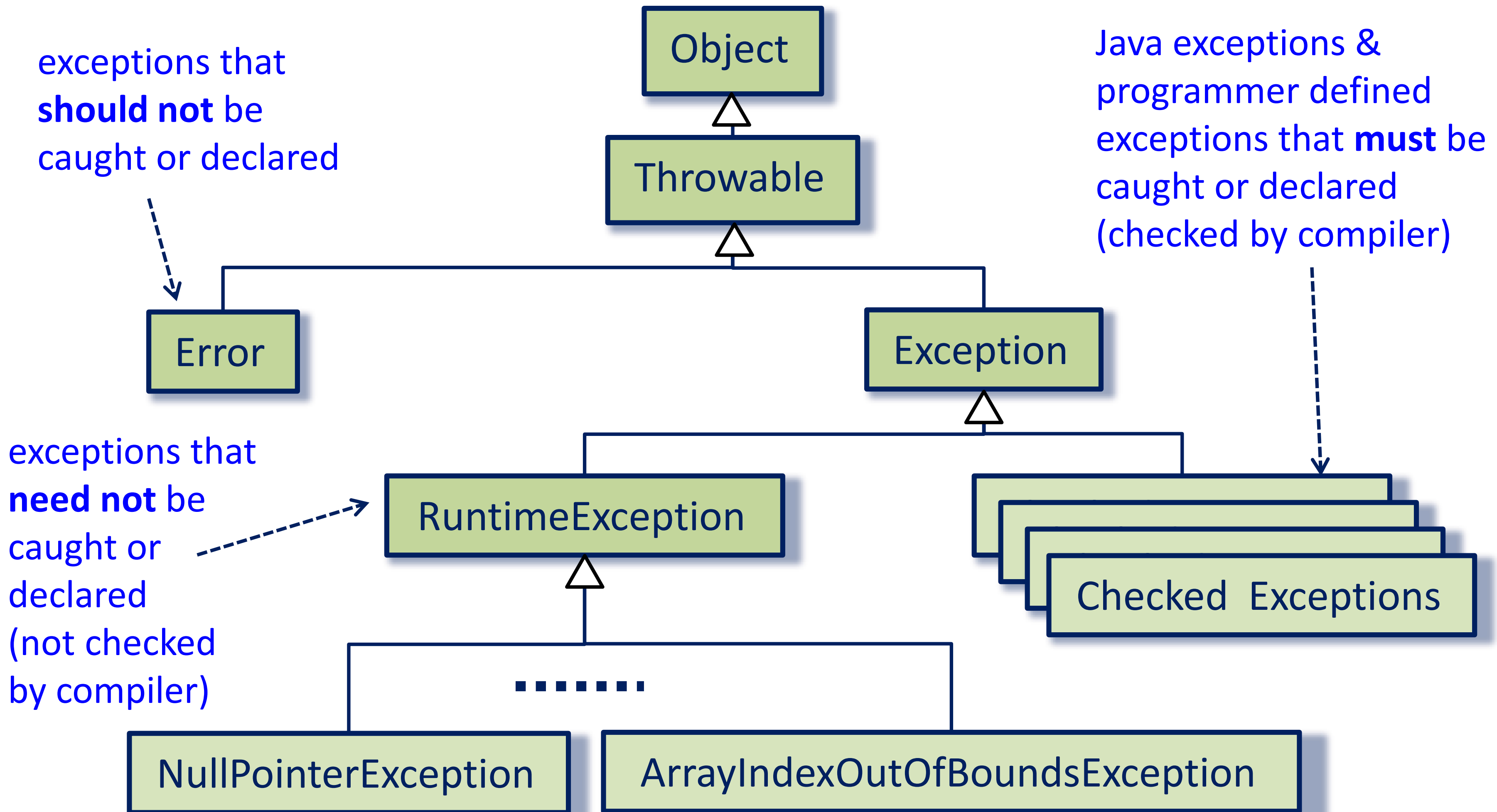
GeometricProgression Subclass

```
public class GeometricProgression extends Progression {

  protected long base;

  /** Constructs progression 1, 2, 4, 8, 16, ... */
  public GeometricProgression( ) { this(2, 1); }          // start at 1 with base of 2

  /** Constructs progression 1, b, b^2, b^3, b^4, ... for base b. */
  public GeometricProgression(long b) { this(b, 1); }          // start at 1

  /** Constructs geometric progression with arbitrary base and start. */
  public GeometricProgression(long b, long start) {
    super(start);
    base = b;
  }

  /** Multiplies the current value by the geometric base. */
  protected void advance( ) {
    current *= base;          // multiply current by the geometric base
  }
}
```

FibonacciProgression Subclass

```java
public class FibonacciProgression extends Progression {

  protected long prev;

  /** Constructs traditional Fibonacci, starting 0, 1, 1, 2, 3, ... */
  public FibonacciProgression( ) { this(0, 1); }

  /** Constructs generalized Fibonacci, with give first and second values. */
  public FibonacciProgression(long first, long second) {
    super(first);
    prev = second - first;        // fictitious value preceding the first
  }

  /** Replaces (prev,current) with (current, current+prev). */
  protected void advance( ) {
    long temp = prev;
    prev = current;
    current += temp;
  }
}
```

Exceptions


Object
Throwable
Error
Exception
RuntimeException
Checked Exceptions
NullPointerException
ArrayIndexOutOfBoundsException
exceptions that **should not** be caught or declared
Java exceptions & programmer defined exceptions that **must** be caught or declared (checked by compiler)
exceptions that **need not** be caught or declared (not checked by compiler)
....

Examples of Exceptions

Exception	Occasion for Use
IllegalArgumentException	Non-null parameter value is inappropriate
IllegalStateException	Object state is inappropriate for method invocation
NullPointerException	Parameter value is null where prohibited
IndexOutOfBoundsException	Index parameter value is out of range
ConcurrentModificationException	Concurrent modification of an object has been detected where it is prohibited
UnsupportedOperationException	Object does not support method

```
public  class  NewException  extends  Exception {
      public NewException() { }                              // no message constructor
      public NewException( String msg ) {super( msg ); }     // detailed message constructor
}
```

Exceptions

- Exceptions are unexpected events that occur during the execution of a program, for example due to:
  - an unavailable resource (error – if not recoverable)
  - unexpected input from a user (checked exception – if recoverable)
  - a logical error on the part of the programmer (run time exception)
- In Java, exceptions are objects that can be **thrown** by code that encounters an unexpected situation.
- An exception may also be **caught** by a surrounding block of code that "handles" the problem.
- If uncaught, an exception causes the virtual machine to stop executing the program and to report an appropriate message to the console.

Catching Exceptions

- The general methodology for handling exceptions is a **try-catch** or **try-catch-finally** construct in which a guarded code fragment that might throw an exception is executed.

```
try {
    guardedBody
} catch (exceptionType1  variable1) {
    remedyBody1
} catch (exceptionType2  variable2) {
    remedyBody2
} finally {
    cleanupBody   // e.g., close file
}
```

- If it **throws** an exception, then that exception is caught by having the flow of control jump to a predefined **catch** block that contains the code to apply an appropriate resolution.
- If no exception occurs in the guarded code, all **catch** blocks are ignored.
- If the **finally** block is present, it is always executed (even if no exception is thrown) and has *higher* precedence than **catch** blocks.

Throwing Exceptions

- Exceptions originate when a piece of Java code finds some sort of problem during execution and throws an exception object.
- This is done by using the **throw** keyword followed by an instance of the exception type to be thrown.
- It is often convenient to instantiate an exception object at the time the exception has to be thrown. Thus, a throw statement is typically written as follows:

  throw new *exceptionType(parameters);*

  where *exceptionType* is the type of the exception and the parameters are sent to that type's constructor.

The **throws** Clause

- When a method is declared, it is possible to explicitly declare, as part of its signature, the possibility that a particular exception type may be thrown during a call to that method.

- The syntax for declaring possible exceptions in a method signature relies on the keyword **throws** (not to be confused with an actual **throw** statement).

- **Example:** the parseInt method of the Integer class has the following formal signature:

  public static int *parseInt(String s)*
  throws *NumberFormatException;*

Design Decision

If an unusual situation occurs, should I throw an exception?

- If you can resolve the unusual situation in a reasonable manner, you likely can use a decision statement instead of throwing an exception.
- If several resolutions to an abnormal occurrence are possible, and you want the client to choose one, you should throw a checked exception.
- If a programmer makes a coding mistake by using your method incorrectly, you can throw a runtime exception. However, you should not throw a runtime exception simply to enable a client to avoid handling it.

Casting

- Casting with Objects allows for conversion between classes and subclasses.
- A **widening conversion** occurs when a type T is converted into a "wider" type U, i.e., "T **IS_A** U"

- **Example:**

  *Super var1 = **new** Sub(...); // implicit widening*

Narrowing Conversions

- A **narrowing conversion** occurs when a type T is converted into a "narrower" type S, i.e., "S **IS_A** T"

- In general, a narrowing conversion of reference types requires an explicit cast.

- **Example:**

```
Super   var1  =   new  Sub(...);      // implicit widening
Sub     var2  =  (Sub)  var1;         // explicit narrowing
```

Quiz: What is the output?

```
public class Quiz {
    static class A {  String a;    public A() { a = "AAA"; }     }
    static class B extends A {  String b;   public B() { b = "BBB"; }   }
    static class C extends B {  String c;   public C() { c = "CCC"; }  }

    public static void main(String[] args) {
        A v = new B(),  w = new C();
        System.out.println( v.a );
        System.out.println( v.b );
        System.out.println( ( (B) v ).b );
        System.out.println( ( (C) v ).a );
        System.out.println( ( (B) w ).a );
    }
}
```

Generics

- Java includes support for writing **generic** types that can operate on a variety of data types while avoiding the need for explicit casts & with type safety through compile-time type-checking.

- Prior to generics (as of Java SE 5), **Object** was used as the universal super-type. Disadvantages:
    - frequent casting to specific actual type.
    - thwarted compiler's type-checking mechanism.

- The generics framework allows us to define a class in terms of a set of **formal type parameters**, undefined at compile time, which can then be used as the declared **non-primitive** type for variables, parameters, and return values within the class definition.

- Those formal type parameters are later specified by **actual type arguments** when using the generic class as a type elsewhere in a program.

Syntax for Generics

- **Example:** a generic paired item by composition:

```
public class Pair<A,B> {
  A first;
  B second;
  public Pair(A a, B b) {                    // constructor
    first = a;
    second = b;
  }
  public A getFirst( ) { return first; }
  public B getSecond( ) { return second;}
}
```

- Can be re-used to instantiate any paired item:
  - Person: (String *name*, Integer *age*)
  - Stock-ticker: (String *stock* , Double *price*)
  - 2D point: (Double *x* , Double *y*)

Type inference with generics
(as of Java SE 7)

1. // declare explicit actual type

```
Pair<String , Double >  bid;
```

2. // instantiate by explicit actual type

```
bid  =  new  Pair<String, Double>("ORCL" , 32.07);
```

Alternatively, rely on **type inference** by **<>** (the "dymond") :

```
// instantiate by type inference
bid = new Pair<> ("ORCL" , 32.07);
```

3. // combined declaration & instantiation:

```
Pair<String , Double >  bid = new Pair<> ("ORCL" , 32.07);
```

4.

```
String   stock  =  bid.getFirst();
double  price  =  bid.getSecond();        // auto unboxing
```

Bounded generics

- **Wild-card "?"** stands for "any class or interface"

- **Bounded generics** with wild-cards:

  <? extends T >
  stands for any subtype of T: any class or interface in the hierarchy rooted at the type represented by the generic type T.

  <? super T >
  stands for any supertype of T: the generic type <T> or higher up in its hierarchy (as direct or indirect super-class or super-interface).

- **Recursive type bounding**:
  e.g., *<T extends Comparable<T> >*
  may be read as: "for any type T that can be compared to itself"

Generics on arrays

- Generics are a compile-time construct
  – the type information is lost at runtime.

- This was a deliberate decision to allow backward compatibility with pre-generics Java code.

- As a consequence:
  - you **can declare** an array of generic type,
  - but you **cannot create** an array of generic type, because the compiler doesn't know how to create an array of an unknown component type.

Generics on arrays

- Two important incompatibilities between arrays & generics:
  - Arrays are **covariant** & **reified**
  - Generics are **invariant** & **non-reified** (aka, type **erase**)
    - **Covariant:** A is subtype of B ⇒ A[ ] is subtype of B[ ]
    - **Invariant:** A & B distinct types ⇒
      List<A> and List<B> have no hierarchical relationship
    - **Reified:** retains & enforces static (compile-time) type at runtime
    - **Non-reified** (aka, type **erase**): loses type information at runtime
- Why is Generic Array Creation not Allowed in Java? Answer.
- So, how can we create "generic" arrays? … next page.

Generics on arrays - 1

```
//  Object-based collection – a prime candidate for generics
//  Raw type is not type-safe:  any type element can be pushed into stack.
public class Stack {
        private Object[ ]   elements;              // declaring raw type array
        private int  size = 0;
        private static final int  CAPACITY = 100;
        public Stack() {                           // raw type constructor
                elements = new Object[CAPACITY];
        }
        public void push (Object e) {
                elements[size++] = e;
        }
        public Object pop() {
                if  (size == 0)  throw new EmptyStackException();
                Object result = elements[--size];
                elements[size] = null;             // eliminate obsolete reference
                return result;
        }
}
```

Generics on arrays - 2

```
// Initial attempt to generify Stack – won't compile!
public class Stack<E> {
        private E[ ]   elements;                    // using "generic" array
        private int  size = 0;
        private static final int  CAPACITY = 100;
        public Stack() {                            // "generic" type constructor
                elements = new E[CAPACITY];   // compiler error
        }
        public void push (E e) {
                elements[size++] = e;
        }
        public E pop() {
                if  (size == 0)  throw new EmptyStackException();
                E result = elements[--size];
                elements[size] = null;              // eliminate obsolete reference
                return result;
        }
}
```

Generics on arrays - 3

```
/**   First solution:   apply explicit cast in constructor.
  *  The elements array will contain only E instances from push(E).
  *  This is sufficient to ensure type safety, but the runtime type
  *  of the reified array won't be E[]; it will always be Object[]!
  */
@SuppressWarnings("unchecked")     // risky – use cautiously!
public Stack() {
        elements = (E[ ]) new Object[CAPACITY];   // explicit cast
}

// … the rest unchanged
```

Generics on arrays - 4

```
/*  Second solution:  apply explicit cast in pop().
 *  Appropriate suppression of unchecked warning
 */
public E pop() {
        if  (size == 0)  throw new EmptyStackException();

        //  push requires elements to be of type E, so cast is correct
        @SuppressWarnings("unchecked")
        E result =  (E)  elements[--size];

        elements[size] = null;              // eliminate obsolete reference
        return result;
}


// ... the rest unchanged
```

Generic Methods

```
public class GenericDemo {
    public static <T> void reverse ( T[ ]  data ) {
        int low = 0 ,  high = data.length – 1;
        while (low < high ) {        // swap data[low] & data[high]
            T temp = data[low];
            data[low++] = data[high];     // post-increment low
            data[high--] = temp;          // post-decrement high
        }
    }
    modifier <T> indicates that this is a generic method
}
```

A call to **reverse(arr)** reverses elements of array **arr** of any declared reference type.

```java
public class GenericDemo {
    public static <T> void reverse(T[] data) {
        int low = 0, high = data.length - 1;
        while (low < high) { // swap data[low] & data[high]
            T temp = data[low];
            data[low++] = data[high]; // post-increment low
            data[high--] = temp; // post-decrement high
        }
    }
    public static <T> void test(T[] data) {
        System.out.println("\nA New Test:");
        for (T e : data)  System.out.print(e + " ");
        System.out.print(" <-- REVERSED --> ");
        reverse(data);
        for (T e : data)  System.out.print(e + " ");
    }
    public static void main(String[] args) {
        test(new Integer[] { 1, 2, 3, 4, 5, 6, 7 });
        test(new String[] { "Merry", "Tom", "Dick", "Harry" });
    }
}
```

Console

<terminated> GenericDemo [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.e

```

A New Test:
1 2 3 4 5 6 7  <-- REVERSED --> 7 6 5 4 3 2 1
A New Test:
Merry Tom Dick Harry  <-- REVERSED --> Harry Dick Tom Merry
```

Bounded wildcards increase API flexibility

- For maximum flexibility, use wildcard types on input parameters that represent **producers** or **consumers**.
- Don't use wildcards on return types.

Motto: PECS stands for producer-extends, consumer-super.

- **Example:** method **max(list)** returns the maximum element of **list**. This needs elements of **list** to be **Comparable** so that we can apply the **compareTo** method on them.
  - max: list is producer, Comparable is consumer.
  - The attempted generic solutions follow …

Bounded wildcards …

```
public static   <E>   E  max( List<E> list)
```

generic E should be Comparable

```
public static   < E extends Comparable<E> >
                  E   max( List<E> list)
```

PECS

```
public static   <E extends Comparable<? super E> >
                  E  max( List<? extends E> list)
```

Bounded wildcards …

```
public static   <E extends Comparable<? super E> >
                E  max( List<? extends E> list)   {
        // see Slide 7 on List ADT & Iterators
        Iterator<? extends E>   iterList  =  list.iterator() ;
        E  result  =  iterList.next() ;
        while  ( iterList.hasNext() ) {
                E e  =  iterList.next() ;
                if  (e.compareTo(result) > 0 )    result  =  e ;
        }
        return result ;
}
```

Nested Classes

- a class definition nested inside the definition of another class.
- There are 4 kinds of nested classes:
  - *static*, *non-static*, *anonymous*, *local*.
    All but the first are called inner classes.

- The main use for nested classes is when defining a class that is strongly affiliated with another class.
  - enhances encapsulation & reduces undesired name conflicts.

- Nested classes are a valuable technique to implement data structures. A nested class can be used to represent a small portion of a larger data structure, or an auxiliary class that helps navigate a primary data structure.

Summary

- **object oriented design principles:**
  abstraction, modularity, encapsulation,
  inheritance, polymorphism
- **program development:**
  design, coding, errors, testing & debugging
- **ADTs**
- **interfaces, concrete & abstract classes**
- **exceptions**
- **casting**
- **generics:** for an in-depth study click [here](#) or read "Effective Java".
- **nested classes**