Abstract classes

- A class with default behaviour:

```
class Creature {
  boolean alive;
  void move()
  {
    System.out.println("Here we go...");
  }
}
```

- An abstract class: subclasses must provide implementation

```
abstract class Creature {
  boolean alive;
  abstract void move();
}
```

Abstract classes

- There cannot be instances of abstract classes.

```
Creature kowe = new Creature(); // Wrong!
//because
kowe.move(); // What would be executed here?
```

- The abstract methods *must* be implemented in the subclasses of an abstract class (unless the subclass itself is also abstract.) This is, there is no default behaviour for an abstract method.

Abstract classes

- An abstract class can have non-abstract methods (which usually represent the "default behaviour" of a method:)

```
abstract class Creature
{
  boolean alive, hungry;
  abstract void move();
  void eat()
  {
    System.out.println("Hmmm...");
    hungry = false;
  }
}
```

Interfaces

- Interfaces are (equivalent to) purely abstract classes, i.e. classes where all the methods are abstract

```
interface Creature
{
  void move();
  void eat();
}
```

is the same as

```
abstract class Creature
{
  abstract void move();
  abstract void eat();
}
```

Interfaces

```
class Human implements Creature
{
  void move()
  {
    System.out.println(“I’m walking...”);
  }
  void eat()
  {
    System.out.println(“I’m eating...”);
  }
  void jump()
  {
    System.out.println(“Up and down...”);
  }
}
```

Abstract classes

```
abstract class Currency
{
  String name;
  String[] countries;
  abstract double exchange_rate();
}
```

Abstract classes

```
class CanadianDollars extends Currency
{
  double exchange_rate()
  {
    return 0.83;
  }
}
```

Abstract classes

```
class Euros extends Currency
{
  double exchange_rate()
  {
    return 1.30;
  }
}
```

Abstract classes

```
class CityList
{
  // ...
  public void sort()
  {
    // Insertion sort
  }
  // ...
}
```

Abstract classes

```
class CityList
{
  // ...
  public void sort()
  {
    // Selection sort
  }
  // ...
}
```

Abstract classes

```
abstract class CityList
{
  // ...
  public abstract void sort();
  // ...
}
```

Abstract classes

```
class CityList_Ins extends CityList
{
  public void sort()
  {
    // Insertion sort
  }
}
```

Abstract classes

```
class CityList_Sel extends CityList
{
  public void sort()
  {
    // Selection sort
  }
}
```

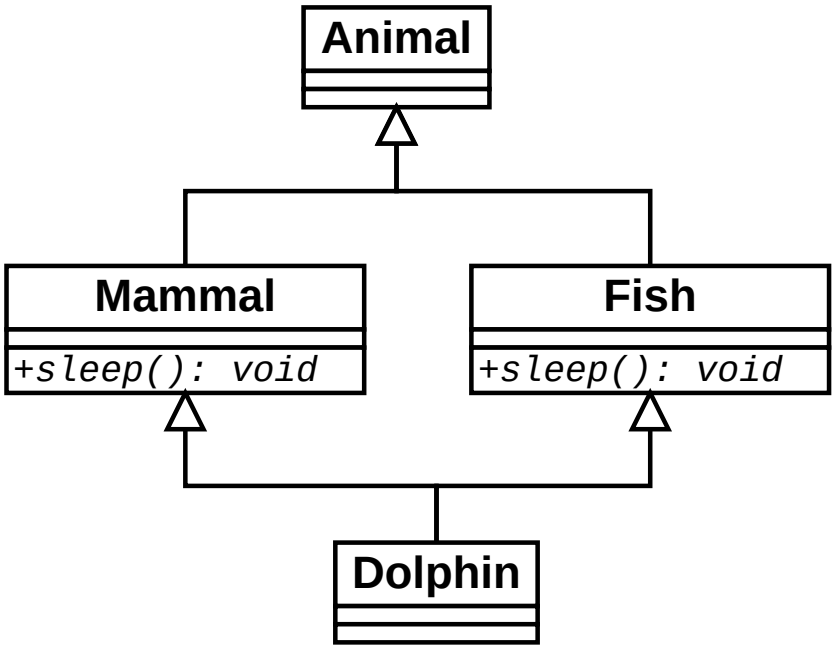
Abstract classes

```
class CityApplication
{
  public static void doSomething()
  {
    CityList l = new CityList_Ins(100);
    sort(l);
  }
  public static void sort(CityList list)
  {
    list.sort();
  }
}
```

Abstract classes

```
class CityApplication
{
  public static void doSomething()
  {
    CityList l = new CityList_Sel(100);
    sort(l);
  }
  public static void sort(CityList list)
  {
    list.sort();
  }
}
```

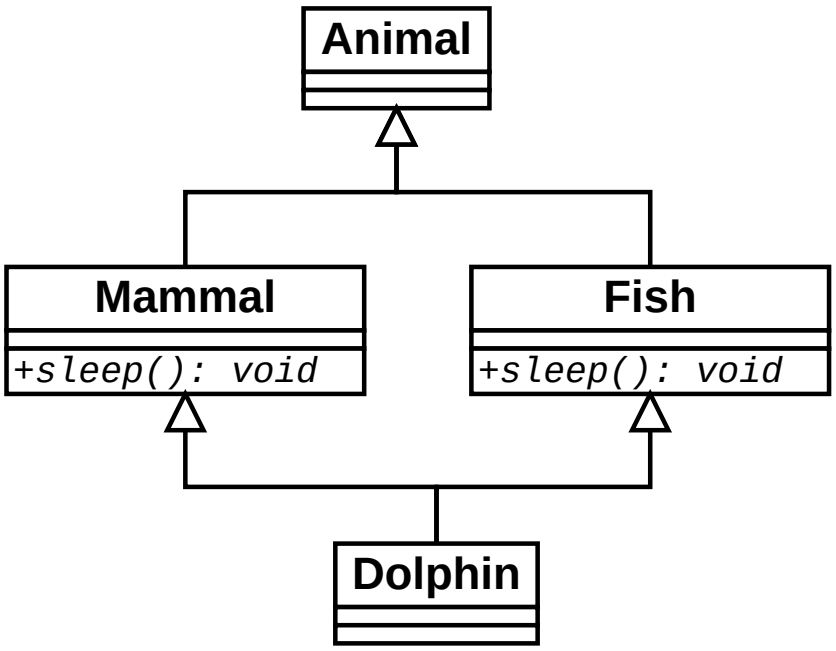
Multiple inheritance




```
class A extends B, C { ... } // Error
```

- Java *does not* support multiple inheritance of classes
- This is because if a method exists in two parents and it is not overriden, then Java cannot know which one to inherit.

Multiple inheritance




```
class A implements B, C { ... } // OK
```

- Java *does* support multiple inheritance of interfaces
- This is because A is forced to provide implementations for all methods in the interfaces, and therefore there is not ambiguity.

Multiple inheritance

- They can be combined:

```
class A extends D implements B, C
{
   ...
}
```

Exception handling

- Errors:
  - Compile-time errors:
    - Syntax errors
    - Typing errors
  - Run-time errors:
    - Logic errors
    - Program crashes

Exception handling

```
int some_method()
{
  int a, b, c, d;
  a = 5;
  b = 0;
  c = a / b;
  d = c + 2;
  return d;
}
```

Exception handling

```
int some_method()
{
  int a, b, c, d;
  a = 5;
  b = 0;
  c = a / b;  // Run-time exception: div by 0
  d = c + 2;
  return d;
}
// ArithmeticException
```

Exception handling

```
int some_method()
{
  int a, b, c, d;
  a = 5;
  b = Keyboard.readInt();
  c = a / b;  // May produce Run-time exception: d
  d = c + 2;
  return d;
}
```

Exception handling

```
int some_method(int b)
{
  int a, c, d;
  a = 5;
  c = a / b;  // May produce Run-time exception: d
  d = c + 2;
  return d;
}
```

Exception handling

```
String some_other_method()
{
  int i;
  String s1 = “hello”, s2;
  char c;
  i = 5;
  c = s1.charAt(i);
  s2 = “the letter is ” + c;
  return s2;
}
// StringIndexOutOfBoundsException
```

Exception handling

```
String some_other_method(int i)
{
  String s1 = “hello”, s2;
  char c;
  c = s1.charAt(i);
  s2 = “the letter is ” + c;
  return s2;
}
```

Exception handling

```
String some_other_method(int i, String s)
{
  String s2;
  char c;
  c = s.charAt(i);
  s2 = “the letter is ” + c;
  return s2;
}
```

Exception handling

```
class Creature {
  void move()
  {
    System.out.println(“Here we go...”);
  }
}

class Zoo {
  void animate(Creature c)
  {
    c.move();
  }
}
```

Exception handling

```
public class ZooTest {
  public static void main(String[] args)
  {
    Zoo my_zoo = new Zoo();
    Creature argos;
    my_zoo.animate(argos);
  }
}
```

Exception handling

```
public class ZooTest {
  public static void main(String[] args)
  {
    Zoo my_zoo = new Zoo();
    Creature argos;
    my_zoo.animate(argos);  // Null-pointer except
  }
}
```

Exception handling

```
class Creature {
  void move()
  {
    System.out.println(“Here we go...”);
  }
}

class Zoo {
  void animate(Creature c)
  {
    if (c != null) c.move();
  }
}
```

Exception handling

```
public class ZooTest {
  public static void main(String[] args)
  {
    Zoo my_zoo = new Zoo();
    Creature argos = new Creature();
    my_zoo.animate(argos);
  }
}
```

Exception handling

```
String some_other_method(int i, String s)
{
  String s2;
  char c;
  c = s.charAt(i);
  s2 = “the letter is ” + c;
  return s2;
}
```

Exception handling

```
String some_other_method(int i, String s)
{
  String s2;
  char c;
  if (s != null && i < s.length()) {
    c = s.charAt(i);
  }
  s2 = “the letter is ” + c;
  return s2;
}
```

Exception handling

```
int some_method(int b)
{
  int a, c, d;
  a = 5;
  c = a / b;  // May produce Run-time exception: d
  d = c + 2;
  return d;
}
```

Exception handling

```
int some_method(int b)
{
  int a, c, d;
  a = 5;
  if (b != 0) c = a / b;
  else c = 0;
  d = c + 2;
  return d;
}
```

Exception handling

- An *exception* is an object that represents a special situation or error that occurs at *runtime*
- If the error or situation occurs, we say that the exception is *raised* or *thrown*.
- An exception may be thrown by the Java Runtime System (JVM) or explicitly by the progrma using the `throw` keyword.
- An exception can be handled by the `try-catch` construct.
- An exception handled by the `try-catch` construct is said to be `caught`.
- Exception objects must be instances of some subclass of `Exception`, or must implement the `Throwable` interface.

Exception handling

- An exception is generated (raised) with the *throw* statement:

  ```
  throw object;
  ```

  where *object* is an instance of a subclass of `Exception` or `Throwable`

- The *try-catch* statement:

  ```
  try {
    statements;
  }
  catch (ExceptionSubclass1 e) {
    statements1;
  }
  catch (ExceptionSubclass2 e) {
    statements2;
  }
  .
  .
  .
  ```

Exception handling

- A try-catch statement executes its default statements in sequence, and
  - If no exception is raised, then computation continues after the catch clauses
  - Otherwise, if an exception is raised, the sequence of statements is interrupted, and execution continues in the catch clause that matches the type of the exception

- After a catch clause finishes, computation continues after the try-catch. This is, the flow of control does not return to the point where the exception occurred. *Note: It never returns to the method that raised the exception, in contrast with a method call.*

- An exception which is not caught by a try-catch, is "propagated", i.e. it is raised again

Exception handling

```
int some_method(int b)
{
  int a, c, d;
  a = 5;
  if (b != 0) c = a / b;
  else c = 0;
  d = c + 2;
  return d;
}
```

Exception handling

```
int some_method(int b)
{
  int a, c, d;
  try {
    a = 5;
    c = a / b;
    d = c + 2;
  }
  catch (ArithmeticException e) {
    d = 2;
  }
  return d;
}
```

Exception handling

```
class Creature {
  void move()
  {
    System.out.println(“Here we go...”);
  }
}

class Zoo {
  void animate(Creature c)
  {
    if (c != null) c.move();
  }
}
```

Exception handling

```
class Creature {
  void move()
  {
    System.out.println(“Here we go...”);
  }
}

class Zoo {
  void animate(Creature c)
  {
    try {
      c.move();
    }
    catch (NullPointerException e) {
    }
  }
}
```

Exception handling

```
class SomeClass {
  static int some_method(int b)
  {
    int a, c, d;
    try {
      a = 5;
      c = a / b;
      d = c + 2;
    }
    catch (ArithmeticException e) {
      d = 2;
    }
    return d;
  }
  static void yet_another_method()
  {
    int x = 5, y;
    y = some_method(x);
    System.out.println(y);
  }
}
```

Exception handling

```
class SomeClass {
  static int some_method(int b)
  {
    int a, c, d;
    try {
      a = 5;
      c = a / b;
      d = c + 2;
    }
    catch (ArithmeticException e) {
      d = 2;
    }
    return d;
  }
  static void yet_another_method()
  {
    int x = 0, y;
    y = some_method(x);
    System.out.println(y);
  }
}
```

Exception handling

```
class SomeClass {
  static int some_method(int b) throws ArithmeticE
  {
    int a, c, d;
    a = 5;
    c = a / b;
    d = c + 2;
    return d;
  }
  static void yet_another_method()
  {
    int x = 5, y;
    try {
      y = some_method(x);
    }
    catch (ArithmeticException e) {
      y = 2;
    }
    System.out.println(y);
  }
}
```

Exception handling

```
class SomeClass {
  static int some_method(int b) throws ArithmeticE
  {
    int a, c, d;
    a = 5;
    c = a / b;
    d = c + 2;
    return d;
  }
  static void yet_another_method()
  {
    int x = 0, y;
    try {
      y = some_method(x);
    }
    catch (ArithmeticException e) {
      y = 2;
    }
    System.out.println(y);
  }
}
```

Exception handling

- Separation of concerns

```
static int q(float f) {
  if (f < 10)
    System.out.println(“Error, f <10, “+f);
  return f * 3 + 1;
}
static void p() {
  float n = Keyboard.readFloat();
  int m = q(n);
  System.out.println(m);
}
```

Exception handling

```
static int q(float f) {
  if (f < 10) {
    System.out.println("Error, f <10, "+f);
    return -1;
  }
  return f * 3 + 1;
}
static void p() {
  float n = Keyboard.readFloat();
  int m = q(n);
  if (m != -1)
    System.out.println(m);
  else
    System.out.println(“Error”);
}
```

Exception handling

```
static int q(float f) {
  if (f < 10) return -1;
  return f * 3 + 1;
}
static void p() {
  float n = Keyboard.readFloat();
  int m = q(n);
  if (m != -1)
    System.out.println(m);
  else
    System.out.println("Error"); //No Error info
}
```

Exception handling

```
static int q(float f) {
  if (f < -5) return -1;
  return f * 3 + 1;
}
static float r(float f) {
  if (f > 15) return -1;
  return f - 2;
}
static void p() {
  float n = Keyboard.readFloat();
  int m = q(r(n));
  if (m != -1)
    System.out.println(m);
  else
    System.out.println("Error"); //No Error info
}
// q(r(13)) = q(11) = 34
// q(r(16)) = q(-1) = -2 // wrong
// q(r(-6)) = q(-8) = -1
```

Exception handling

```
static int q(float f) {
  if (f < -5) return -1;
  return f * 3 + 1;
}
static float r(float f) {
  if (f > 15) return -1;
  return f - 2;
}
static void p() {
  float n = Keyboard.readFloat();
  int partial1 = r(n);
  if (partial1 == -1)
    System.out.println(“Error in r”);
  else {
    int partial2 = q(partial1);
    if (partial2 == -1)
      System.out.println("Error in q"); //No Error
    else
      System.out.println(partial2);
  }
}
```

Exception handling

```
class MyException extends Exception {
  String message;
  MyException(String m)
  {
    message = m
  }
  public String toString()
  {
    return “MyException occurred: “+message;
  }
}
```

Exception handling

```
static int q(float f) throws MyException
{
  if (f < -5)
    throw new MyException(“q: ”+f);
  return f * 3 + 1;
}
static float r(float f) throws MyException
{
  if (f > 15)
    throw new MyException(“r: ”+f);
  return f - 2;
}
```

Exception handling

```
static void p() {
  float n = Keyboard.readFloat();
  try {
    int m = q(r(n));
    System.out.println(m);
  }
  catch (MyException e) {
    System.out.println(e);
  }
}
```

Exception handling

```
class MyException extends Exception {
  String message;
  MyException(String m) { message = m; }
  public String toString()
  {
    return “MyException occurred: ”+message;
  }
}

class MyOtherException extends Exception {
  int code;
  MyOtherException(int c) { code = c; }
  public String toString() { return “”+code; }
}
```

Exception handling

```
static int q(float f) throws MyOtherException
{
  if (f < -5)
    throw new MyOtherException(7);
  return f * 3 + 1;
}
static float r(float f) throws MyException
{
  if (f > 15)
    throw new MyException(“r: ”+f);
  return f - 2;
}
```

Exception handling

```
static void p()
{
  float n = Keyboard.readFloat();
  try {
    int m = q(r(n));
    System.out.println(m);
  }
  catch (MyException e) {
    System.out.println(e);
  }
  catch (MyOtherException e) {
    String s = e;
  }
}
```

Exception handling

- If MyOtherException is not caught, it repropagates

```
static void p() throws MyOtherException
{
  float n = Keyboard.readFloat();
  try {
    int m = q(r(n));
    System.out.println(m);
  }
  catch (MyException e) {
    System.out.println(e);
  }
}
```

- Note: p does not throw an exception explicitely

The end