Polymorphism

```
class A
{
  void m() { ... }
}

class B extends A
{
  void m() { ... }
}

class C extends A
{
  void m() { ... }
}
```

Polymorphism

```
class D
{
  void q(A obj)
  {
    obj.m();
  }
  void p()
  {
    A obj;
    obj = new B();
    q(obj);
    obj = new C();
    q(obj);
  }
}
```

Polymorphism

```
abstract class A
{
  abstract void m();
}

class B extends A
{
  void m() { ... }
}

class C extends A
{
  void m() { ... }
}
```

Polymorphism

```
class D
{
  void q(A obj)
  {
    obj.m();
  }
  void p()
  {
    A obj;
    obj = new B();
    q(obj);
    obj = new C();
    q(obj);
  }
}
```

Polymorphism

```
interface A
{
  void m();
}

class B implements A
{
  void m() { ... }
}

class C implements A
{
  void m() { ... }
}
```

Polymorphism

```
class D
{
  void q(A obj)
  {
    obj.m();
  }
  void p()
  {
    A obj;
    obj = new B();
    q(obj);
    obj = new C();
    q(obj);
  }
}
```

Polymorphism

```
interface A
{
  void m();
}

class B implements A
{
  void m() { ... }
  void f() { ... }
}

class C implements A
{
  void m() { ... }
  void g() { ... }
}
```

Polymorphism

```
class D
{
  void q(A obj)
  {
    obj.m();
    if (c instanceof B)
    {
      B h = (B)c;
      h.f();
    }
    if (c instanceof C)
    {
      C m = (C)c;
      m.g();
    }
  }
  void p() { ... }
}
```

Casting object references

```
class Zoo
{
  void animate(Creature c)
  {
    c.move();
    if (c instanceof Human)
    {
      Human h = (Human)c;
      h.jump();
    }
    if (c instanceof Martian)
    {
      Martian m = (Martian)c;
      m.hop();
    }
  }
}
```

Generic Programming

- Why use polymorphism?
- Many methods execute the same *algorithm*, but differ only in the *data-types* of its arguments
- Generic programming:
  - Defining methods that can handle different data-types of arguments
  - Defining methods that are independent of the data-types of the arguments

Generic Programming

```
int linear_search(int[] a, int value)
{
  int index = 0;
  while (index < a.length)
  {
    if (value == a[index])
    {
      return index;
    }
    index++;
  }
  return -1;  // Not found
}
```

Generic Programming

```
public int book_index(Book[] list, String title)
{
  int i;
  i = 0;
  while (i < list.length)
  {
    Book m = list[i];
    if (m != null)
    {
      String s = m.title();
      if (s.equals(title))
      {
        return i;
      }
    }
    i++;
  }
  return -1;
}
```

Generic Programming

```
public interface EqualityComparable
{
  public boolean equals(EqualityComparable other);
}
```

Generic Programming

```
public class Book implements EqualityComparable
{
  private String title, author;
  public Book(String t, String d)
  {
    title = t;
    author = d;
  }
  public boolean equals(EqualityComparable other)
  {
    if (other instanceof Book)
    {
      Book b = (Book)other;
      String t = b.title;
      return title.equals(t);
    }
    return false;
  }
}
```

Generic Programming

```
public interface EqualityComparable
{
  public boolean equals(Object other);
}
```

- `Object` is the parent class of all classes

Generic Programming

```
public class Book implements EqualityComparable
{
  private String title, author;
  public Book(String t, String d)
  {
    title = t;
    author = d;
  }
  public boolean equals(Object other)
  {
    if (other instanceof Book)
    {
      Book b = (Book)other;
      String t = b.title;
      return title.equals(t);
    }
    return false;
  }
}
```

Generic Programming

```
public class Student implements EqualityComparable
{
  private String name, id;
  public Student(String n, String i)
  {
    name = n;
    id = i;
  }
  public boolean equals(Object other)
  {
    if (other instanceof Student)
    {
      Student b = (Student)other;
      return name.equals(b.name)
          && id.equals(b.id);
    }
    return false;
  }
}
```

Generic Programming

```
public interface EqualityComparable
{
  public boolean equals(Object other);
}
```

- `Object` is the parent class of all classes

Generic Programming

```
int generic_linear_search(EqualityComparable[] list
                          EqualityComparable targe
{
  int i;
  i = 0;
  while (i < list.length)
  {
    EqualityComparable item = list[i];
    if (item != null)
    {
      if (item.equals(target))
      {
        return i;
      }
    }
    i++;
  }
  return -1;
}
```

Generic Programming

```
int generic_linear_search(EqualityComparable[] list
                          EqualityComparable targe
{
  int i;
  i = 0;
  while (i < list.length)
  {
    EqualityComparable item = list[i];
    if (item != null)
    {
      if (item.equals(target)) // Dynamic-dispatch
      {
        return i;
      }
    }
    i++;
  }
  return -1;
}
```

Generic Programming

```
public class SearchAlgorithms
{
  int generic_linear_search(EqualityComparable[] li
                            EqualityComparable targ
  {
    int i;
    i = 0;
    while (i < list.length)
    {
      EqualityComparable item = list[i];
      if (item != null)
      {
        if (item.equals(target))
        {
          return i;
        }
      }
      i++;
    }
    return -1;
  }
```

```
}
```

Generic Programming

```
import SearchAlgorithms;
public class LibraryApplication
{
  public static void main(String[] args)
  {
    Book[] list = ...;
    Book b = new Book(“Hamlet”);
    int i;
    i = SearchAlgorithms.generic_linear_search(list, b);
    // ...
  }
}
```

Generic Programming

```
import SearchAlgorithms;
public class LibraryApplication
{
  public static void main(String[] args)
  {
    Student[] list = ...;
    Student b = new Student(“Hamlet”, “260098128”)
    int i;
    i = SearchAlgorithms.generic_linear_search(lis
    // ...
  }
}
```

Generic Programming

```
public interface Comparable
{
  public int compareTo(Object other);
}
```

- `Object` is the parent class of all classes

Generic Programming

```
void insertion_sort(Book[] list)
{
  int i, j;
  String key;
  Book focus;
  j = 1;
  while (j < list.length)
  {
    focus = list[j];
    key = focus.title();
    i = j - 1;
    while (i >= 0 &&
      key.compareTo(list[i].title()) < 0 )
    {
      list[i+1] = list[i];
      i--;
    }
    list[i+1] = focus;
    j++;
  }
}
```

Generic Programming

```
void insertion_sort(Comparable[] list)
{
  int i, j;
  Comparable focus;
  j = 1;
  while (j < list.length)
  {
    focus = list[j];
    i = j - 1;
    while (i >= 0 &&
      focus.compareTo(list[i]) < 0 )
    {
      list[i+1] = list[i];
      i--;
    }
    list[i+1] = focus;
    j++;
  }
}
```

Generic Programming

```
public class Book implements EqualityComparable,
                             Comparable
{
  private String title, author;
  public Book(String t, String d)
  { ... }
  public boolean equals(Object other)
  { ... }
  public int compareTo(Object other)
  {
    if (other instanceof Book)
    {
      Book b = (Book)other;
      return title.compareTo(b.title);
    }
  }
}
```

Generic Programming

```
public class Student implements EqualityComparable
                                Comparable
{
  private String name, id;
  public Book(String n, String i)
  { ... }
  public boolean equals(Object other)
  { ... }
  public int compareTo(Object other)
  {
    if (other instanceof Student)
    {
      Student b = (Student)other;
      return name.compareTo(b.name);
    }
  }
}
```

Generic Programming

```
class Library
{
  private Book[] book_list;
  public int next_available;

  public Library(int max_capacity) { ... }

  public int number_of_books() { ... } // Accessor

  public void add_book(Book m) { ... } // Mutator

  public int book_index(String title) { ... } // A

  public void delete_book(String title) { ... } //
} // End of Library
```

Generic Programming

```
class List
{
  private Object[] array;
  public int next_available;

  public List(int max_capacity) { ... }

  public int size() { ... } // Accessor

  public void add(Object m) { ... } // Mutator

  public int indexOf(Object target) { ... } // Acc

  public void remove(int target) { ... } // Mutato
} // End of Library
```

Generic Programming

Classes from the standard library

- ArrayList
- Vector
- LinkedList

Generic Programming

```
import java.util.ArrayList;
public class Test
{
  public static void main(String[] args)
  {
    ArrayList list = new ArrayList();
    list.add(“Tomatos”);
    list.add(“Cheese”);
    list.add(“Basil”);
    int i = list.indexOf(“Cheese”);
    list.remove(i);
  }
}
```

Object Oriented Programming

- The execution of an OO program consists of
  - Creation of objects
  - Interaction between objects (message-passing)
- Defining features of an OO language:
  - Class definitions (describing the types of objects and their structure,)
  - Object instantiation (creation,)
  - Message-passing (invoking methods,)
  - Aggregation (object structure, *has-a* relationships)
  - Encapsulation (objects as abstract units, hiding,)
  - Inheritance,
  - Polymorphism

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 = scanner.nextInt();
  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 program 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 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) 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 Food {
  boolean fresh, smelly;
}
class FoulSmell extends Exception {
  public String toString() {
    return “Yuck”;
  }
}
class FoodPoison extends Exception {
  public String toString() {
    return “Ouch”;
  }
}
```

Exception handling

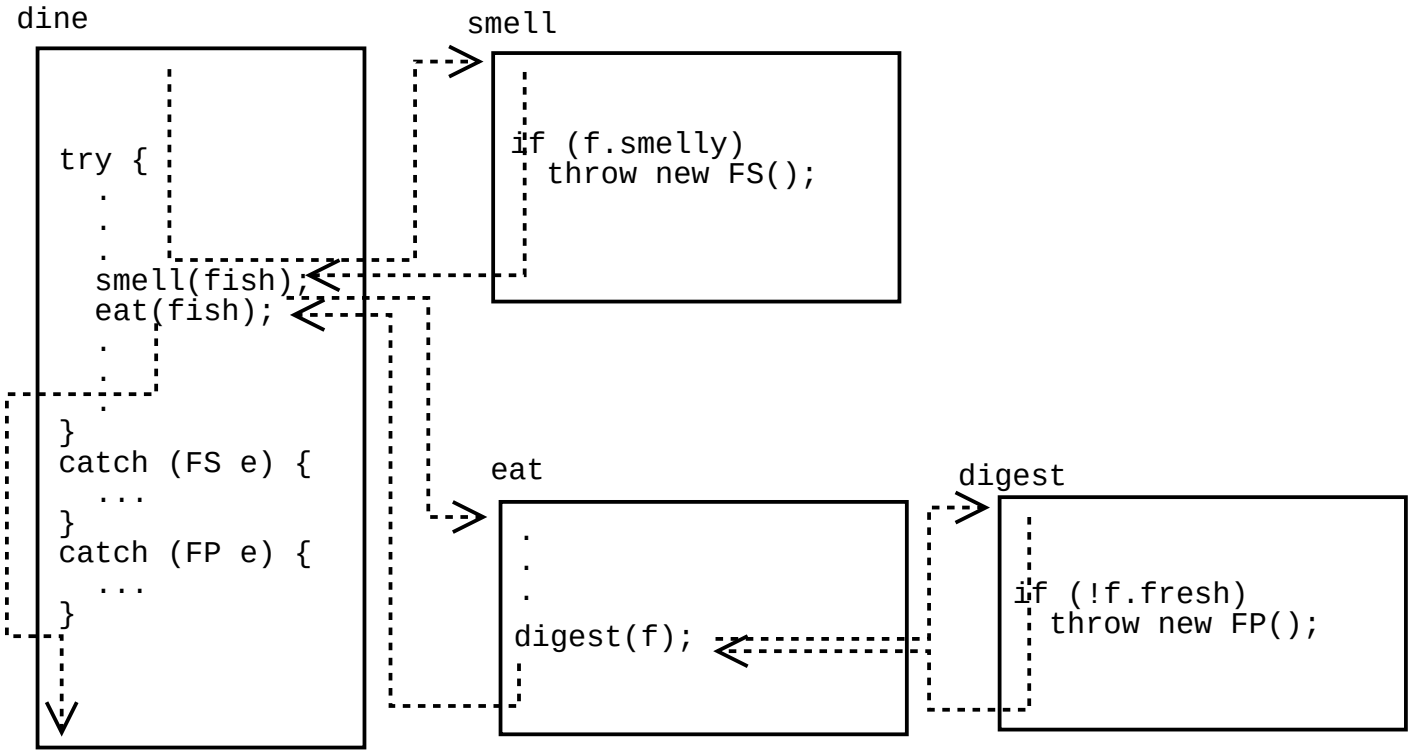
```
static void smell(Food f) throws FoulSmell
{
  if (f.smelly)
    throw new FoulSmell();
  System.out.println("Smells OK");
}
static void eat(Food f) throws FoodPoison
{
  System.out.println("Hmmm...");
  digest(f);
}
static void digest(Food f) throws FoodPoison
{
  if (!f.fresh)
    throw new FoodPoison();
}
```

Exception handling

```
static void dine()
{
  try {
    Food fish = new Food();
    fish.smelly = false;
    fish.fresh = true;
    smell(fish);
    eat(fish);
  }
  catch (FoulSmell e) {
    System.out.println(e);
  }
  catch (FoodPoison e) {
    System.out.println(e);
  }
}
```

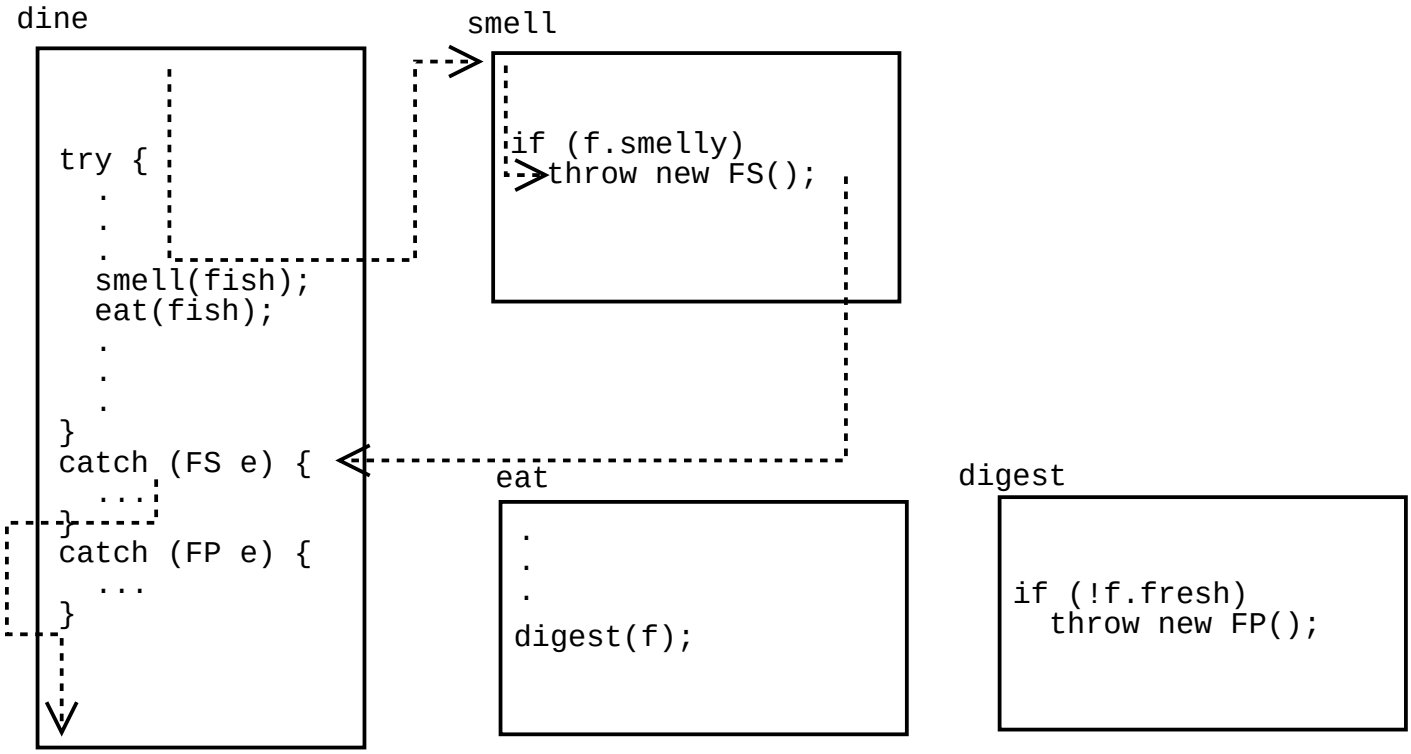
Exception handling

```
fish.smelly = false; fish.fresh = true;
```



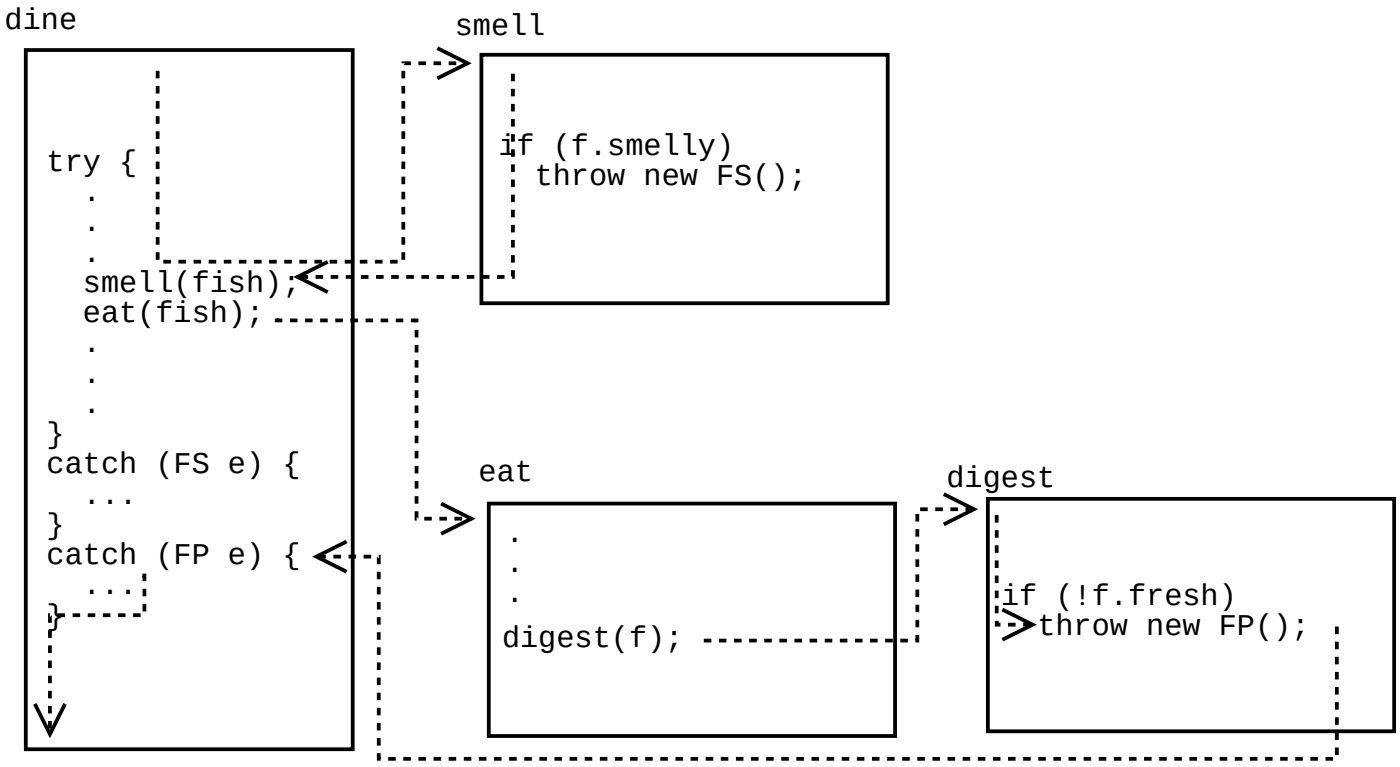
Exception handling

```
fish.smelly = true;
```



Exception handling

```
fish.smelly = false; fish.fresh = false;
```



Exception handling

- A method can throw more than one class of exceptions:

```
void m() throws A, B, ...
{
   ... throw new A() ...
   ... throw new B(...) ...
}
```

- ... but the exception needs not be raised explicitly in the method itself: it can be raised by another method called by `m`.

Exception handling

- Exceptions can be used not only for errors, but for control-flow too:

```
class Sheep {
  private int id;
  public Sheep(int i) { n = i; }
  public void jump()
  {
    System.out.println(“Sheep #”+id+” jumped”);
    if (id == 6)
      throw new LoudSound(i);
  }
}
```

Exception handling

```
class LoudSound extends Throwable {
  private int n;
  public LoudSound(int i) { n = i; }
  public String toString()
  {
    return “I was in sheep #”’+n;
  }
}
```

Exception handling

```
class GoToSleep {
  public static void main(String[] args)
  {
    try {
      for (int i = 1; i < 100; i++) {
        Sheep s = new Sheep(i);
        s.jump();
      }
      System.out.println("zzzz...");
    }
    catch (LoudSound s) {
      System.out.println(s);
    }
  }
}
```

Exception handling

- Some exceptions arise without an explicit throw.
- Some standard exceptions

```
Exception
  RunTimeException
    IndexOutOfBounds
    StringIndexOutOfBounds
    ArithmeticException (e.g. division by 0)
    NullPointerException
  NoSuchMethodException
  ClassNotFoundException
```

The end