CS571: Programming Languages

Subtype Principle

- In any operation that expects an object of type T, it is acceptable to supply object of type T', which is subtype of T.

Subtype Principle (Java, subtype3/)

- Example (Java):

```
public class A
{ public void f1() {System.out.println("f1");}}
public class B extends A
{ public void f2() {System.out.println("f2");}}
public class Main {
    public static void main(String args[]){
        A i = new A(); B s = new B();
        i.f1();
        s.f1();
        s.f2();
        i.f2(); }}
```


Subtype Principle (Java, subtype3/)

- Example (Java):

```
public class A
{ public void f1() {System.out.println("f1");}}
public class B extends A
{ public void f2() {System.out.println("f2");}}
public class Main {
    public static void main(String args[]){
        A i = new A(); B s = new B();
        i.f1();            //legal
        s.f1();
        s.f2();
        i.f2(); }}
```


Subtype Principle (Java, subtype3/)

- Example (Java):

```
public class A
{ public void f1() {System.out.println("f1");}}
public class B extends A
{ public void f2() {System.out.println("f2");}}
public class Main {
    public static void main(String args[]){
        A i = new A(); B s = new B();
        i.f1();        //legal
        s.f1();        //legal – subtype principle
        s.f2();
        i.f2(); }}
```


Subtype Principle (Java, subtype3/)

- Example (Java):

```
public class A
{ public void f1() {System.out.println("f1");}}
public class B extends A
{ public void f2() {System.out.println("f2");}}
public class Main {
    public static void main(String args[]){
        A i = new A(); B s = new B();
        i.f1();        //legal
        s.f1();        //legal – subtype principle
        s.f2();        //legal
        i.f2(); }}
```

Subtype Principle (Java, subtype3/)

- Example (Java):

```
public class A
{  public void f1() {System.out.println("f1");}}

public class B extends A
{  public void f2() {System.out.println("f2");}}

 public class Main {
    public static void main(String args[]){
        A i = new A(); B s = new B();
       i.f1();        //legal
       s.f1();        //legal – subtype principle
       s.f2();        //legal
       i.f2(); }}     //illegal
```

7

Subtype Principle (subtype.cpp)

Does subtype principle always hold in C++?

8

Subtype Principle (subtype.cpp)

Does subtype principle always hold in C++?

```
class A
{ public: void f();
  …};
class B: private A
{…}; // no public redefinition of f

A x;
B y;
x.f();
y.f();
```


9

Subtype Principle (subtype.cpp)

Does subtype principle always hold in C++?

```
class A
{ public: void f();
  …};
class B: private A
{…}; // no public redefinition of f

A x;
B y;
x.f();       //legal
y.f();
```


10

Subtype Principle (subtype.cpp)

Does subtype principle always hold in C++?

```
class A
{ public: void f();
  …};
class B: private A
{…}; // no public redefinition of f

A x;
B y;
x.f();       //legal
y.f();       //illegal – subtype principle does not hold
```


11

Subtype Principle (java)

Does subtype principle always hold in java?

Yes, java supports only public inheritance

12

Upcasting vs. Downcasting

Upcasting

- Upcasting: casting an object to a more general type - always legal in both C++ and Java.

Example (Java):

```
class Mammal { }

class Cat extends Mammal { }

Cat c = new Cat();
Mammal m = c; // upcasting
```


Downcasting

- Downcasting: casting an object to a more specific type - Java inserts a run-time check to ensure that the cast is legal.

Downcasting (Java)

A: super class B: sub class

```
A a;
B b;
a = new A();
b = a;
```


Downcasting (Java)

A: super class B: sub class

```
A a;
B b;
a = new A();
b = a;     // a compile-time error in Java
```


Downcasting (Java)

A: super class B: sub class

```
A a;
B b;
a = new A();
b = (B) a;
```

Downcasting (Java)

A: super class B: sub class

```
A a;
B b;
a = new A();
b = (B) a;    // now no compiler error – downcasting
              // java.lang.ClassCastException
```


Downcasting (Java)

A: super class B: sub class

```
A a;
B b;
a = new B();
b = (B) a;
```


Downcasting (Java)

A: super class B: sub class

```
A a;
B b;
a = new B();
b = (B) a;    //no compile and run-time error
```


Static and Dynamic Method Binding

Static/Dynamic Method Binding

Example (Java):

```
SuperClass superClass1 = new SubClass();

superClass1.f();  //f() in SubClass() or SuperClass()?
```


Static vs. Dynamic Methods

- Static/Dynamic binding: decide which method gets executed at compile/run time
- It is possible to offer both static and dynamic binding of methods in an object-oriented language.
  - C++: dynamic binding is an option, but is not the default.
    - Virtual function: dynamic
    - Non-virtual: static
  - Java: all methods are implicitly virtual, i.e., dynamic binding always applies. Static binding can be achieved by appropriate use of the super keyword.

Dynamic Binding (Java, Dynamic/)

```
public class A
{ public A()
  { System.out.println("A"); }
  public void f()
   { System.out.println("A.f");}
}
class B extends A
{ public B() { super(); }
  public void f()
   { System.out.println("B.f");}
}
```

```
A r = new B;
r.f();
```


Dynamic Binding (Java, Dynamic/)

```
public class A
{ public A()
  { System.out.println("A"); }
  public void f()
   { System.out.println("A.f");}
}
class B extends A
{ public B() { super(); }
  public void f()
   { System.out.println("B.f");}
}
```

```
A r = new B;
r.f();
```

Output: A
B.f

Dynamic Binding (Java, Dynamic3/)

```
class A
{ void p() {System.out.println("A.p");}
  void q() {System.out.println("A.q");}
  void f() {p(); q();}
}
class B extends A
{ void p() {System.out.println("B.p");}
  void q() {System.out.println("B.q"); super.q();}
  void f() {p(); q();}
}
public class Main{
  public static void main(String[] args)
   {A a = new A(); a.f();
    a = new B(); a.f();
   }
}
```


Dynamic Binding (Java, Dynamic3/)

```
class A
{ void p() {System.out.println("A.p");}
  void q() {System.out.println("A.q");}
  void f() {p(); q();}
}
class B extends A
{ void p() {System.out.println("B.p");}
  void q() {System.out.println("B.q"); super.q();}
  void f() {p(); q();}
}
public class Main{
  public static void main(String[] args)
   {A a = new A(); a.f();
    a = new B(); a.f();
   }
}
```

Output:
A.p
A.q
B.p
B.q
A.q

Dynamic Binding (Java)

```
class A
{ void p() {System.out.println("A.p");}
  void q() {System.out.println("A.q");}
  void f() {p(); q();}
}
class B extends A
{ void p() {System.out.println("B.p");}
  void q() {System.out.println("B.q"); super.q();}
}
public class Main{
  public static void main(String[] args)
   {A a = new A(); a.f();
    a = new B(); a.f();
   }
}
```


Dynamic Binding (Java)

```
class A
{ void p() {System.out.println("A.p");}
  void q() {System.out.println("A.q");}
  void f() {p(); q();}
}
class B extends A
{ void p() {System.out.println("B.p");}
  void q() {System.out.println("B.q"); super.q();}
}
public class Main{
  public static void main(String[] args)
   {A a = new A(); a.f();
    a = new B(); a.f();
   }
}
```

Output:
A.p
A.q
B.p
B.q
A.q

Static/Dynamic Method Binding (virtual10.cpp)

```
class A
{ public:
     void p() {cout << "A::p\n";}
     virtual void q() {cout <<
   "A::q\n";}
   };
class B : public A
{ public:
     void p() { cout << "B::p\n";}
     void q() { cout << "B::q\n";}
};
```

```
int main()
{  A* a1 = new B;
   a1 -> p();
   a1 -> q();
}
```


Static/Dynamic Method Binding (virtual10.cpp)

```
class A
{ public:
     void p() {cout << "A::p\n";}
     virtual void q() {cout <<
   "A::q\n";}
   };
class B : public A
{ public:
     void p() { cout << "B::p\n";}
     void q() { cout << "B::q\n";}
};
```

```
int main()
{  A* a1 = new B;
   a1 -> p();
   a1 -> q();
}
```

Result:
A::p
B::q

Virtual Function

- Once Virtual, always virtual
  - Once a base-class defines a function as virtual, it remains virtual through out the inheritance hierarchy.
- Costs 10% to 20% extra overhead compared to a non-virtual function call.

Static/Dynamic Method Binding (virtual.cpp)

```
class A
{ public:
     void p() {cout << "A::p\n";}
     void q() {cout << "A::q\n";}
     void f() { p(); q();}
 };
class B : public A
{ public:
     void p() { cout << "B::p\n";}
     void q() { cout << "B::q\n";}
};
```

```
int main()
{ A a; B b;
  a.f(); b.f();
  a = b; a.f();
}
```


Static/Dynamic Method Binding (virtual.cpp)

```
class A
{ public:
     void p() {cout << "A::p\n";}
     void q() {cout << "A::q\n";}
     void f() { p(); q();}
 };
class B : public A
{ public:
     void p() { cout << "B::p\n";}
     void q() { cout << "B::q\n";}
};
```

```
int main()
{ A a; B b;
  a.f(); b.f();
  a = b; a.f();
}
```

Result:
A::p
A::q
A::p
A::q
A::p
A::q

Static/Dynamic Method Binding (virtual.cpp)

```
class A
{ public:
     void p() {cout << "A::p\n";}
     virtual void q() {cout <<
   "A::q\n";}
     void f() { p(); q();}
 };
class B : public A
{ public:
     void p() { cout << "B::p\n";}
     void q() { cout << "B::q\n";}
};
```

```
int main()
{ A a; B b;
  a.f(); b.f();
  a = b; a.f();
}
```

Static/Dynamic Method Binding (virtual.cpp)

```
class A
{ public:
    void p() {cout << "A::p\n";}
    virtual void q() {cout <<
  "A::q\n";}
    void f() { p(); q();}
 };
class B : public A
{ public:
    void p() { cout << "B::p\n";}
    void q() { cout << "B::q\n";}
};
```

```
int main()
{ A a; B b;
  a.f(); b.f();
  a = b; a.f();
}
```

```
Result:
A::p
A::q
A::p
B::q
A::p
A::q
```


Static/Dynamic Method Binding (virtual1.cpp)

```
class A
{ public:
    void p() {cout << "A::p\n";}
    virtual void q() {cout <<
  "A::q\n";}
    void f() { p(); q();}
 };
class B : public A
{ public:
    void p() { cout << "B::p\n";}
    void q() { cout << "B::q\n";}
};
```

```
int main()
{ A* a = new A;
  B* b = new B;
  a -> f(); b -> f();
  a = b; a -> f();
}
```


Static/Dynamic Method Binding (virtual1.cpp)

```
class A
{ public:
    void p() {cout << "A::p\n";}
    virtual void q() {cout <<
  "A::q\n";}
    void f() { p(); q();}
 };
class B : public A
{ public:
    void p() { cout << "B::p\n";}
    void q() { cout << "B::q\n";}
};
```

```
int main()
{ A* a = new A;
  B* b = new B;
  a -> f(); b -> f();
  a = b; a -> f();
}
```

```
Result:
A::p
A::q
A::p
B::q
A::p
B::q
```


Overloading vs. Parametric Polymorphism

Overloading vs. Parametric Polymorphism

- Parameteric Polymorphism: the same function body is reused to deal with different data types.
- Overloading: no reuse of the overloaded function since there is in fact a different function body corresponding to each argument type.

Overloading

- Overloading (aka adhoc Polymorphism): same function name is used to represent different functions, each of which may take arguments of a specific type.
  - Operator '+' is overloaded in most languages so that they can be used to add integers or reals.
    - How does a translator disambiguate these two use of the "+" symbol.
      - 2 + 3 – integer addition
      - 2.0 + 3.0 – floating-point addition
      - 2 + 3.0 – most languages convert 2 to 2.0

Overloading (Cont.)

- One name has multiple (overloaded) meanings within a particular namespace.
  - Function Overloading
  - Operator Overloading
- C++ and Ada allow extensive overloading of both function names and operators.
- Java also allows overloading, but only on function names, not operators

Overload Resolution

- We must have a way of determining which function declaration should be used for a particular use of an overloaded function name.
  - not only check the name of a function, but also check the number of its parameters and their data types.
- This process of choosing a unique function among many with the same name is called overload resolution.

Example of Function Overloading (Overload1.java)

```
public class Overload {
   public static int max(int x, int y)
  { return x > y ? x : y;}

   public static double max(double x, double y)
      { return x > y ? x : y;}

   public static int max(int x, int y, int z)
      { return max(max(x,y),z);}

   public static void main(String[] args)
      { System.out.println(max(1,2));
        System.out.println(max(1,2,3));
        System.out.println(max(4,1.3));
      }}
```

count the number of parameters and then look to see if the parameter values are integers or doubles

Example of Function Overloading (Overload1.java)

```
public class Overload {
   public static int max(int x, int y)
  { return x > y ? x : y;}
   public static double max(double x, double y)
      { return x > y ? x : y;}

   public static int max(int x, int y, int z)
      { return max(max(x,y),z);}

   public static void main(String[] args)
      { System.out.println(max(1,2));       //2
        System.out.println(max(1,2,3));     //3
        System.out.println(max(4,1.3));     //4.0
  }
}
```

count the number of parameters and then look to see if the parameter values are integers or doubles

Function Overloading (Overload1.java)

```
public class Overload {
   public static int max(int x, int y)
  { return x > y ? x : y;}
   public static double max(double x, double y)
      { return x > y ? x : y;}

   public static int max(int x, int y, int z)
      { return max(max(x,y),z);}

  }
```

- max(4,1.3) – which max?

Function Overloading (Overload1.java)

```
public class Overload {
   public static int max(int x, int y)
  { return x > y ? x : y;}
   public static double max(double x, double y)
      { return x > y ? x : y;}

   public static int max(int x, int y, int z)
      { return max(max(x,y),z);}

  }
```

- max(4,1.3) – which max?
  - Java: permits conversions that do not lose information. 4 → 4.0.

Function Overloading (overload.cpp)

```
#include <iostream>
using namespace std;
class Overload {
    public:
    int add(int x, int y) { return x+y;}
    double add(double x, double y) { return x+y;}
    int add(int x, int y, int z) { return x+y+z;}
};
int main (int argc, char **argv) {
     Overload o;
     cout << o.add(1,2) << endl;
     cout << o.add(4,1.3) << endl;
      cout << o.add(4.1,1) << endl;
}
```


Function Overloading (overload.cpp)

error: call of overloaded `add(int, double)' is ambiguous

error: call of overloaded `add(double, int)' is ambiguous

Function Overloading (overload.cpp)

```
#include <iostream>
using namespace std;
class Overload {
    public:
    int add(int x, int y) { return x+y;}
    double add(double x, double y) { return x+y;}
    int add(int x, int y, int z) { return x+y+z;}
};
int main (int argc, char **argv) {
     Overload o;
     cout << o.add(1,2) << endl;
     cout << o.add((double)4,1.3) << endl;
      cout << o.add((int)4.1,1) << endl;
}
```


Function Overloading (overload.cpp)

```
#include <iostream>
using namespace std;
class Overload {
    public:
    int add(int x, int y) { return x+y;}
    double add(double x, double y) { return x+y;}
    int add(int x, int y, int z) { return x+y+z;}
};
int main (int argc, char **argv) {
     Overload o;                                    Output:
     cout << o.add(1,2) << endl;                      3
     cout << o.add((double)4,1.3) << endl;            5.3
      cout << o.add((int)4.1,1) << endl;              5
}
```


Operator Overloading (operator.cpp)

```
#include <iostream>
using namespace std;
typedef struct { int i; double d;} IntDouble;
bool operator < (IntDouble x, IntDouble y)
{ return x.i < y.i && x.d < y.d; }
IntDouble operator + (IntDouble x, IntDouble y)
{   IntDouble z;
    z.i = x.i + y.i;
    z.d = x.d + y.d;
    return z;  }
int main()
{   IntDouble x = {1,2.1}, y = {5,3.4};
    if (x < y) x = x  + y;
    else y = x + y;
    cout << x.i << " " << x.d << endl;
    return 0; }
```


Operator Overloading (operator.cpp)

```
#include <iostream>
using namespace std;
typedef struct { int i; double d; } IntDouble;
bool operator < (IntDouble x, IntDouble y)
{ return x.i < y.i && x.d < y.d; }
IntDouble operator + (IntDouble x, IntDouble y)
{   IntDouble z;
    z.i = x.i + y.i;
    z.d = x.d + y.d;
    return z;  }
int main()
{   IntDouble x = {1,2.1}, y = {5,3.4};
    if (x < y) x = x  + y;
    else y = x + y;
    cout << x.i << " " << x.d << endl;     //6 5.5
    return 0; }
```

Parametric Polymorphism

- The same function works for arguments of different types.
- C has templates, Java does not support templates.

Function Templates (C++)

```
template <typename T> void swap (T *a, T *b)
{
   T temp;
   temp = *a;
   *a = *b;
   *b = temp;
}
```

- swap: function template,
- T: a dummy type that will be filled in by the compiler
- a, b and temp are of type T

Function Templates (template1.cpp)

```
#include <iostream>
using namespace std;
int main ( )
{
   int a = 5, b = 6;
   float c = 7.6, d = 9.8;
   char e = 'M', f = 'Z';
   swap (&a, &b);          // replace T with int
   swap (&c, &d);          // replace T with float
   swap (&e, &f);          // replace T with char
   cout << "a=" << a << " and b=" << b << endl;
   cout << "c=" << c << " and d=" << d << endl;
   cout << "e=" << e << " and f=" << f << endl;
}
```

Output: a=6 and b=5
c=9.8 and d=7.6
e='Z' and f='M'

Abstract Methods and Classes

Final classes & methods in Java

- A final class cannot have subclasses. All methods in a final class are implicitly final.

  public final class MyFinalClass {...}

- A final method cannot be overridden by subclasses.
  - E.g. prevent unexpected behavior from a subclass altering a method that may be important to the function of the class.

```
public class MyClass {
      public final void myFinalMethod() {...}
}
```


Final Variables in Java

- A variable may only be assigned a value once.
  - The value of a final variable is not necessarily known at compile time.

```
public class Sphere
{  public static final double PI = 3.1415926;
   public final double radius;
       public final double xpos;
       public final double ypos;
       public final double zpos;
       Sphere(double x, double y, double z, double r)
       { radius = r; xpos = x; ypos = y; zpos = z; }
}
```

Abstract Methods and Classes (Java)

- An abstract method is a method that is declared without an implementation

```
public abstract double area();
```

- If a class includes abstract methods, the class must be declared abstract.

```
public abstract class ClosedFigure
{ public abstract double area();
    …… }
```


Abstract Methods and Classes (Java)

- Abstract classes cannot be instantiated, but they can be subclassed.
- When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class.
  - Otherwise, the subclass must also be declared abstract.

Abstract Methods and Classes (Java)

```
public abstract class ClosedFigure
{ public abstract double area();
  ……
}
```

- It cannot be given a general implementation of area for ClosedFigure, but an implementation will be deferred until a subclass is defined with enough properties to enable the area to be determined.
  - Abstract methods are called deferred.
- Defining abstract methods makes it a requirement that every subclass of ClosedFigure has an area function.

Example: ClosedFigure

```
public abstract class ClosedFigure {
    protected double x,y;
    public void whatPlace() {
        System.out.println("My position: "+x+","+y);
    }
    public abstract double area();
}
```


Example: ClosedFigure

- Circle and Square are sub-classes of ClosedFigure
  - Inherit variables x and y, and method whatPlace().
  - Implement method area()

```
class Circle extends ClosedFigure {
    protected double radius;
    public Circle(double r) {radius=r;}
    public double area() {return(radius*radius*3.1415);}
}
class Square extends ClosedFigure{
    protected double edge;
    public Square(double e){edge=e;}
    public double area() {return(edge*egde);}
}
```


Abstract Classes & Pure Virtual Functions (C++)

- Pure virtual function: A class is made abstract by declaring one or more of its virtual functions to be pure by placing "= 0" in its declaration.

  E.g. `virtual void draw() = 0;`

  - "= 0": pure specifier.
  - Do not provide the implementation

Abstract Classes & Pure Virtual Functions (C++)

- Every concrete subclass must override all pure virtual functions in the base class with concrete implementations.
- If not overridden, the sub-class will also be abstract.

Interface

Interface

- An interface may only contain methods with empty bodies and constant declarations (variables declarations which are declared to be both static and final).

```
interface Y
{    void f()
     ……
}
```


Interface

- Interface cannot be instantiated – they can only be implemented by classes or extended by other interfaces.
- If a class that implements an interface does not implement all of the interface methods, provided that the class is declared to be abstract.

```
abstract class X implements Y
{ // implements all but one method of Y }
class XX implements Y
{ // implements all methods in Y }
```


Interface

- A human and parrot can both whistle.
- It would not make sense to represent Humans and Parrots as subclasses of a Whistler class,
- Rather they would most likely be subclasses of an Animal class, but would both implement the Whistler interface.

Abstract Classes vs Interfaces

- Unlike interfaces, abstract classes can contain fields that are not static and final, and they can contain implemented methods.
- If an abstract class contains only abstract method declarations, it should be declared as an interface instead.

References (Cont.)

- Abstract methods and classes: http://java.sun.com/docs/books/tutorial/java/IandI/abstract.html
- Interface: http://java.sun.com/docs/books/tutorial/java/IandI/createinterface.html

Section 10.8 Implementation issues in OO

Allocation for Data Members

- Data members are allocated next to each other
- E.g. Java program

```
class Point {
        private double x, y;
}
```

space for x
space for y

Allocation for Data Members (Cont.)

- Data members for a derived class immediately follow the data members of the base class

E.g. Java:

```
class Point{
  public double x,y; }
```

```
class ColoredPoint extends Point{
  public Color color;}
```

Space for x
Space for y

Space for x
Space for y
Space for color

- The instance variables x and y inherited by any ColorPoint object from Point object can be found at the same offset from the beginning of its allocated space as for any point object

Allocation for Functions (C++)

- Non-virtual functions:
  - Not allocated inside objects
  - The location of such functions are known at load time, as is the case for ordinary functions in an imperative language.

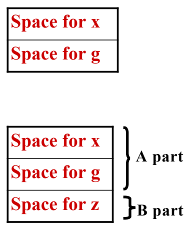
Implementation of Virtual Function (C++)

- Virtual Function: the method to use for a call is not known except during execution
  - Solution 1
    - Treat function members like data members.
    - Allocate storage for them within the object.

Implementation of Virtual Function (C++)

Solution 1:

```
class A:{
    public:
        double x;
        void f() {...};
        virtual void g() {...};
};
class B: public A{
    public:
        double z;
        void f() {...};
        virtual void g() {...};
};
```

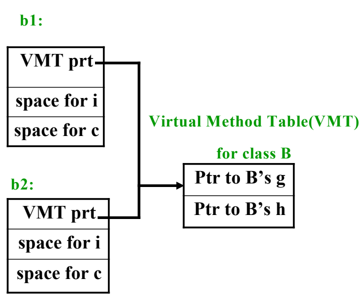


Implementation of Virtual Function (C++)

- Disadvantage of solution 1
  - Each object structure must maintain a list of all virtual functions available at that moment – the list could be very large
- Solution 2
  - Maintain a table of virtual methods for each class at some central location that can be statically loaded and have a single pointer to this table stored in each object structure
  - Such a table is called a virtual method table (VMT)

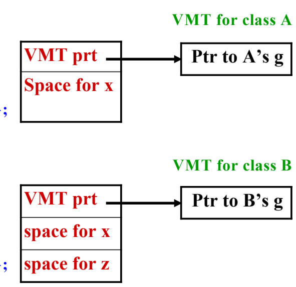
Implementation of Virtual Function (C++)

```
class B {
    int i;
    char c;
    virtual void g();
    virtual void h();
}
B b1, b2;
```



Implementation of Virtual Function (C++)

```
class A:{
    public:
        double x;
        void f() {...};
        virtual void g() {...};
};
class B: public A{
    public:
        double z;
        void f() {...};
        virtual void g() {...};
};
```



Exception

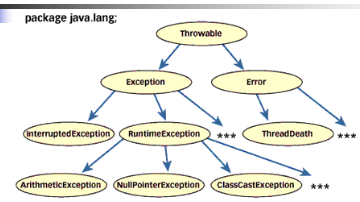
Exception

- Exception is used to indicate that an abnormal condition has occurred.
- When errors or unusual conditions arise during the execution of a function (e.g., divide by zero, null pointer access), control is transferred to a handler for the exception.
- Throw: a statement is said to throw an exception if the execution of this statement leads to an exception.
- When a method encounters an abnormal condition that it cannot handle itself, it may throw an exception.
- Catch: a catch statement is used in C++/Java to declare a handler.

Exception (Java)

- In Java, exceptions are objects. You can throw only those objects whose classes descend from Class Throwable.
  - Throwable serves as the base class for an entire family of classes, declared in java.lang, that your program can initiate and throw.

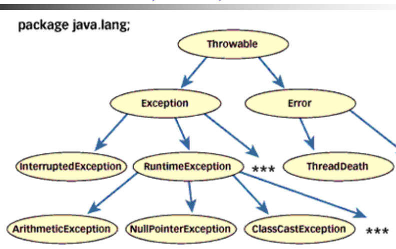
Throwable (Java)



http://www.javaworld.com/javaworld/jw-07-1998/jw-07-exceptions.html

- Exceptions are thrown to signal abnormal conditions that can often be handled by some catcher.
- Errors are usually thrown for more serious problems, e.g. OutOfMemoryError.
- In general, code you write should throw only exceptions, not errors.

Throwable (Java)



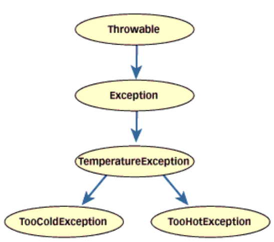
- You can throw objects of your own design - declare the corresponding class as a subclass of some member of the Throwable family, usually extends class Exception.

Catch an Exception (Java)

- To catch an exception in Java, write a try block with one or more catch clauses.
  - The try block contains the code that may throw an exception
  - Each catch clause specifies one exception type that it is prepared to handle.
  - If the code in the try block throws an exception, the associated catch clauses will be examined by the Java virtual machine.
  - If the virtual machine finds a catch clause that can be used to handle the thrown exception, the program continues execution starting with the first statement of that catch clause.

Example: Exception (Java)

- Simulates a customer of a virtual cafe drinking a cup of coffee
- Exceptional conditions that might occur while the customer sips.



Example: Exception (Java)

- If the coffee is cold, throw a TooColdException. If the coffee is overly hot, throw a TooHotException.

TemperatureException extends Exception { }
TooColdException extends TemperatureException { }
TooHotException extends TemperatureException { }

- **Throw an exception**

throw new TooColdException();

Example: Exception (Java)

```
class VirtualPerson
  { private static final int tooCold = 65;
    private static final int tooHot = 85;
    public void drinkCoffee(CoffeeCup cup)
        throws TooColdException, TooHotException {
          int temperature = cup.getTemperature();
          if (temperature <= tooCold) { throw new TooColdException(); }
  else if (temperature >= tooHot) { throw new TooHotException(); } } }

class CoffeeCup {
  // 75 degrees Celsius: the best temperature for coffee
  private int temperature = 75;
  public void setTemperature(int val) { temperature = val; }
  public int getTemperature() { return temperature; }
}
```


Example: Exception (Java)

- If a method isn't prepared to catch the exception, it throws the exception up to its calling method, and so on.
- drinkCoffee() method throws a TooColdException
  - Because the exception isn't caught by drinkCoffee(), the Java virtual machine examines its calling method, i.e., the main() method, to see if it has a catch clause prepared to handle the exception.

Example: Exception (Java)

```
import java.io.*;
public class Main {
  public static void main(String args[]){
    CoffeeCup coffee = new CoffeeCup();
    coffee.setTemperature(50);
    VirtualPerson person = new VirtualPerson();
    try{person.drinkCoffee(coffee);}
    catch(TooColdException e) {
      System.out.println("Coffee is too cold.");
      // Deal with an irate customer...
    }
    catch (TooHotException e) {
      System.out.println("Coffee is too hot.");
      // Deal with an irate customer...
    }
  }
}
```


Exceptions in C++

- C++
  - Earlier version of C++ did not support exception handling
  - Even now, most C++ programs do not use exceptions
  - Exceptions may not be objects
- Java
  - Exceptions were included from the beginning.
  - Java programs crash much less frequently due to unchecked errors as compared to C++ programs.
  - Exceptions are objects whose classes descend from Class Throwable

Example: Exception Handling (C++)

```
#include <iostream>
using namespace std;
int fac(int n) {
    if (n <= 0) throw (-1);
    else if (n >= 15) throw (15);
    else return n*fac(n-1);};
int main() {
    int k;
    try {k=fac(-1);}
    catch(int i) {
        if (i == -1) cout << "negative value invalid";
        else cout << "n is greater than 15"; }
    catch (...) {cout << "unknown exception";}
};
```

fac(-1) - print "negative value invalid"

Example: Exception Handling (C++)

```
#include <iostream>
using namespace std;
int fac(int n) {
    if (n <= 0) throw (-1);
    else if (n > 15) throw (15);
    else return n*fac(n-1);};
int main() {
    int k;
    try {k=fac(16);}
    catch(int i) {
        if (i == -1) cout << "negative value invalid";
        else cout << "n is greater than 15"; }
    catch (...) {cout << "unknown exception";}
};
```

fac(16) - print "n is greater than 15"

If an unexpected error arises when evaluating fac, then "unknown exception" will be printed.

Summary: C++ vs Java

C++ vs Java

- Year developed
  - C++: developed by AT&T Bell Lab in 1979
  - Java: developed by Sun Microsystems in 1990
- Compile/interpret
  - C++: compiled
  - Java: both compiled and interpreted
- Speed
  - Java: much slower than C++
  - C++: 10-20 times faster than equivalent java code. Most operating systems are written using C/C++

C++ vs Java

- Overloading
  - C++: both function and operator overloading
  - Java: function overloading
- Inheritance
  - C++: support multiple inheritance
  - Java: does not support multiple inheritance, but support multiple interface inheritance
  - C++: support private/protected inheritance
  - Java: support only public inheritance

C++ vs Java

- Access level
  - Java: has package access level
  - C++: does not have package access level
- Pointers
  - C++: support pointers
  - Java: does not support pointers
- Deallocation
  - C++: destructor
  - Java: garbage collection

C++ vs Java

- Constants
  - C++: const
  - Java: final + static
- Structure/union
  - C++: support structure/union
  - Java: does not support
- Interface
  - Java supports interface
  - The interface concept is not supported in C++

C++ vs Java

- Method binding
  - C++: static method binding (default)
  - Java: dynamic method binding (default)
- Array bound checking
  - C++: no array bound checking
  - Java: has array bound checking
- Multi-threaded programming
  - Java: easy to write multi-threaded programs
  - C++: harder to write multi-threaded programs