

Inheritance (Part 2)

Preconditions and Inheritance

- ▶ precondition
 - ▶ what the method assumes to be true about the arguments passed to it
- ▶ inheritance (is-a)
 - ▶ a subclass is supposed to be able to do everything its superclasses can do
- ▶ how do they interact?

Strength of a Precondition

- ▶ to strengthen a precondition means to make the precondition more restrictive

```
// Dog setEnergy
// 1. no precondition
// 2. 1 <= energy
// 3. 1 <= energy <= 10
public void setEnergy(int energy)
{ ... }
```



weakest precondition

strongest precondition

Preconditions on Overridden Methods

- ▶ a subclass can change a precondition on a method *but it must not strengthen the precondition*
- ▶ a subclass that strengthens a precondition is saying that it cannot do everything its superclass can do

```
// Dog setEnergy
// assume non-final
// @pre. none

public
void setEnergy(int nrg)
{ // ... }
```

```
// Mix setEnergy
// bad : strengthen precond.
// @pre. 1 <= nrg <= 10

public
void setEnergy(int nrg)
{
    if (nrg < 1 || nrg > 10)
    { // throws exception }
    // ...
}
```

-
- ▶ client code written for **Dogs** now fails when given a **Mix**

```
// client code that sets a Dog's energy to zero
public void walk(Dog d)
{
    d.setEnergy(0);
}
```

- ▶ remember: a subclass must be able to do everything its ancestor classes can do; otherwise, clients will be (unpleasantly) surprised

Postconditions and Inheritance

- ▶ postcondition
 - ▶ what the method promises to be true when it returns
 - ▶ the method might promise something about its return value
 - "returns size where size is between 1 and 10 inclusive"
 - ▶ the method might promise something about the state of the object used to call the method
 - "sets the size of the dog to the specified size"
 - ▶ the method might promise something about one of its parameters
- ▶ how do postconditions and inheritance interact?

Strength of a Postcondition

- ▶ to strengthen a postcondition means to make the postcondition more restrictive

```
// Dog getSize
// 1. no postcondition
// 2. 1 <= this.size
// 3. 1 <= this.size <= 10
public int getSize()
{ ... }
```



Postconditions on Overridden Methods

- ▶ a subclass can change a postcondition on a method *but it must not weaken the postcondition*
- ▶ a subclass that weakens a postcondition is saying that it cannot do everything its superclass can do

```
// Dog getSize
//
// @post. 1 <= size <= 10
```

```
public
int getSize()
{ // ... }
```

```
// Dogzilla getSize
// bad : weaken postcond.
// @post. 1 <= size
```

```
public
int getSize()
{ // ... }
```

Dogzilla: a made-up breed of dog that has no upper limit on its size

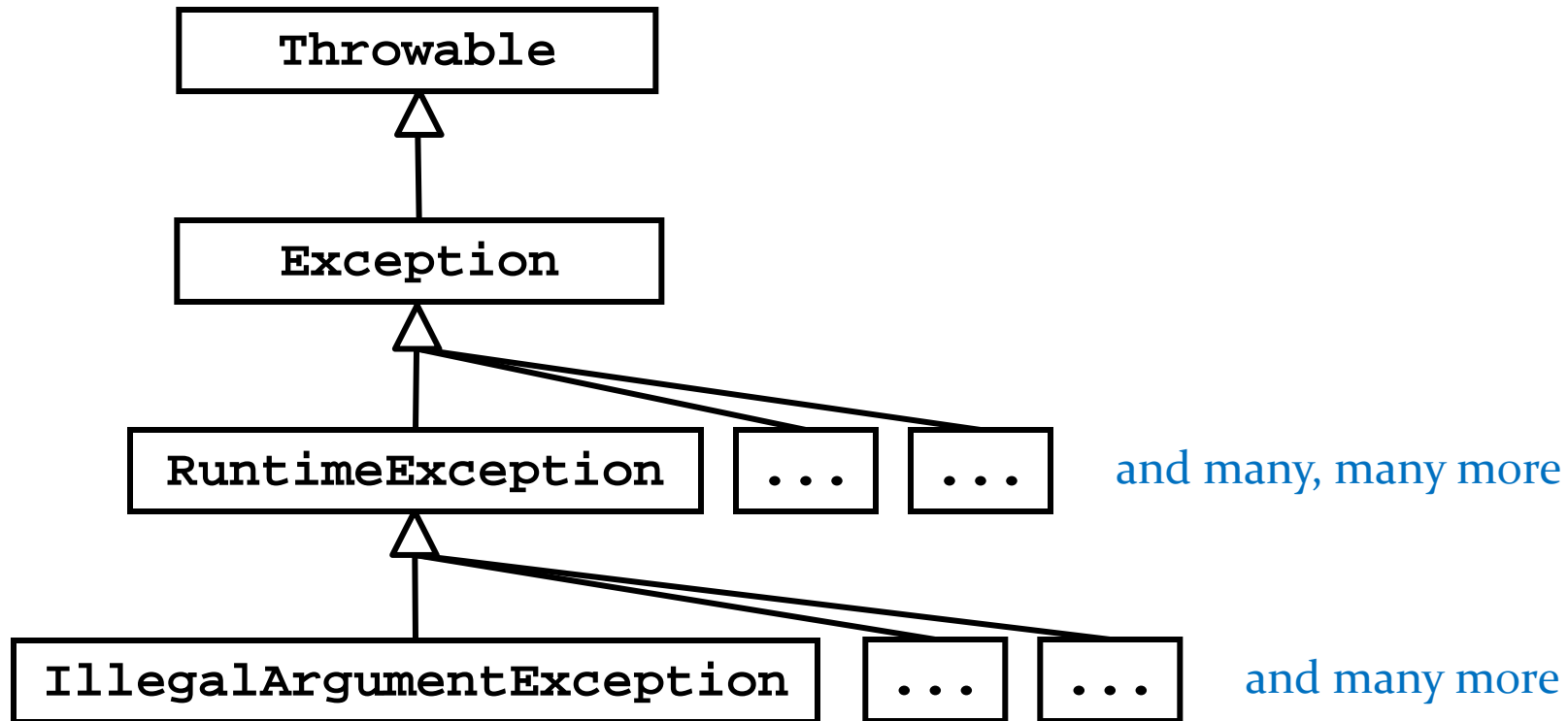
-
- ▶ client code written for **Dogs** can now fail when given a **Dogzilla**

```
// client code that assumes Dog size <= 10
public String sizeToString(Dog d)
{
    int sz = d.getSize();
    String result = "";
    if (sz < 4)          result = "small";
    else if (sz < 7)    result = "medium";
    else if (sz <= 10) result = "large";
    return result;
}
```

- ▶ remember: a subclass must be able to do everything its ancestor classes can do; otherwise, clients will be (unpleasantly) surprised

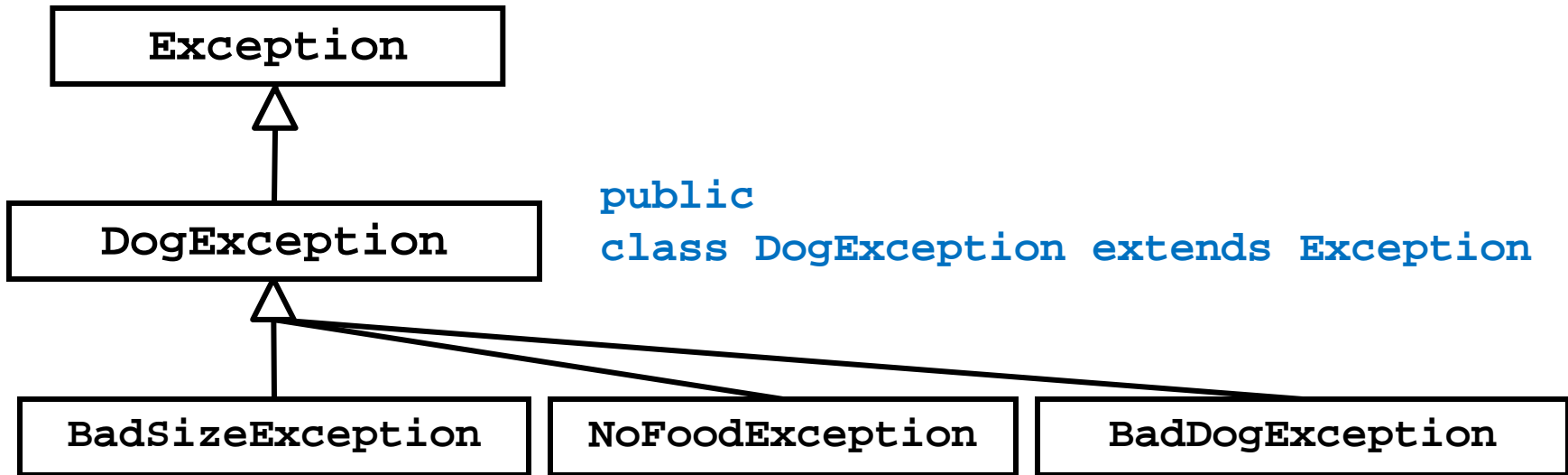
Exceptions

- ▶ all exceptions are objects that are subclasses of `java.lang.Throwable`



User Defined Exceptions

- ▶ you can define your own exception hierarchy
 - ▶ often, you will subclass Exception



Exceptions and Inheritance

- ▶ a method that claims to throw a *checked* exception of type **X** is allowed to throw any checked exception type that is a subclass of **X**
 - ▶ this makes sense because exceptions are objects and subclass objects are substitutable for ancestor classes

```
// in Dog
public void someDogMethod() throws DogException
{
    // can throw a DogException, BadSizeException,
    //                NoFoodException, or BadDogException
}
```

-
- ▶ a method that overrides a superclass method that claims to throw a checked exception of type **X** can also claim to throw a checked exception of type **X** or a subclass of **X**
 - ▶ remember: a subclass is substitutable for the parent type

```
// in Mix
@Override
public void someDogMethod() throws DogException
{
    // ...
}
```

Which are Legal?

► in Mix

@Override

public void someDogMethod() throws BadDogException



@Override

public void someDogMethod() throws Exception



@Override

public void someDogMethod()



@Override

public void someDogMethod()
throws DogException, IllegalArgumentException



Review

1. Inheritance models the _____ relationship between classes.
2. Dog is a _____ of Object.
3. Dog is a _____ of Mix.
4. Can a Dog instance do everything a Mix instance can?
5. Can a Mix instance do everything a Dog instance can?
6. Is a Dog instance substitutable for a Mix instance?
7. Is a Mix instance substitutable for a Dog instance?

-
8. Can a subclass use the private fields of its superclass?
 9. Can a subclass use the private methods of its superclass?
 10. Suppose you have a class X that you do not want anyone to extend. How do you enforce this?
 11. Suppose you have an immutable class X. Someone extends X to make it mutable. Is this legal?
 12. What do you need to do to enforce immutability?

-
13. Suppose you have a class Y that extends X.
- a. Does each Y instance have a X instance inside of it?
 - b. How do you construct the X subobject inside of the Y instance?
 - c. What syntax is used to call the superclass constructor?
 - d. What is constructed first—the X subobject or the Y object?
 - e. Suppose Y introduces a brand new method that needs to call a public method in X named xMethod. How does the new Y method call xMethod?
 - f. Suppose Y overrides a public method in X named xMethod. How does the overriding Y method call xMethod?

-
14. Suppose you have a class Y that extends X. X has a method with the following precondition:

`@pre. value must be a multiple of 2`

If Y overrides the method which of the following are acceptable preconditions for the overriding method:

- a. `@pre. value must be a multiple of 2`
- b. `@pre. value must be odd`
- c. `@pre. value must be a multiple of 2 and must be less than 100`
- d. `@pre. value must be a multiple of 10`
- e. `@pre. none`

-
14. Suppose you have a class Y that extends X. X has a method with the following postcondition:

`@return - A String of length 10`

If Y overrides the method which of the following are acceptable postconditions for the overriding method:

- a. `@return - A String of length 9 or 10`
- b. `@return - The String "weimaraner"`
- c. `@return - An int`
- d. `@return - The same String returned by toString`
- e. `@return - A random String of length 10`



15. Suppose Dog toString has the following Javadoc:

```
/*  
 * Returns a string representation of a dog.  
 * The string is the size of the dog followed by a  
 * a space followed by the energy.  
 * @return The string representation of the dog.  
 */
```

Does this affect subclasses of Dog?

Inheritance Recap

- ▶ inheritance allows you to create subclasses that are substitutable for their ancestors
 - ▶ inheritance interacts with preconditions, postconditions, and exception throwing
- ▶ subclasses
 - ▶ inherit all non-private features
 - ▶ can add new features
 - ▶ can change the behaviour of non-final methods by *overriding* the parent method
 - ▶ contain an instance of the superclass
 - ▶ subclasses must construct the instance via a superclass constructor

Puzzle 3

- ▶ Write the class **Enigma**, which extends **Object**, so that the following program prints false:

```
public class Conundrum
{
    public static void main(String[] args)
    {
        Enigma e = new Enigma();
        System.out.println( e.equals(e) );
    }
}
```

- ▶ You must not override **Object.equals()**

[*Java Puzzlers* by Joshua Block and Neal Gaffer]

Polymorphism

- ▶ inheritance allows you to define a base class that has fields and methods
 - ▶ classes derived from the base class can use the public and protected base class fields and methods
- ▶ polymorphism allows the implementer to change the behaviour of the derived class methods

```
// client code
public void print(Dog d) {
    System.out.println( d.toString() );
}
                                Dog toString
                                CockerSpaniel toString
                                Mix toString

// later on...
Dog          fido = new Dog();
CockerSpaniel lady = new CockerSpaniel();
Mix          mutt = new Mix();
this.print(fido);
this.print(lady);
this.print(mutt);
```


-
- ▶ notice that **fido**, **lady**, and **mutt** were declared as **Dog**, **CockerSpaniel**, and **Mutt**
 - ▶ what if we change the declared type of **fido**, **lady**, and **mutt** ?

```
// client code
public void print(Dog d) {
    System.out.println( d.toString() );
}
    Dog toString
    CockerSpaniel toString
    Mix toString

// later on...
Dog        fido = new Dog();
Dog        lady = new CockerSpaniel();
Dog        mutt = new Mix();
this.print(fido);
this.print(lady);
this.print(mutt);
```

-
- ▶ what if we change the `print` method parameter type to `Object` ?

```
// client code
public void print(Object obj) {
    System.out.println( obj.toString() );
}
    Dog toString
    CockerSpaniel toString
    Mix toString
// later on...
    Date toString
Dog        fido = new Dog();
Dog        lady = new CockerSpaniel();
Dog        mutt = new Mix();
this.print(fido);
this.print(lady);
this.print(mutt);
this.print(new Date());
```

Late Binding

- ▶ polymorphism requires *late binding* of the method name to the method definition
- ▶ late binding means that the method definition is determined at run-time

non-static method

`obj.toString()`

run-time type of
the instance `obj`

Declared vs Run-time type

```
Dog lady = new CockerSpaniel( );
```

declared
type

run-time or actual
type



-
- ▶ the **declared type** of an instance determines what methods can be used

```
Dog lady = new CockerSpaniel();
```

- ▶ the name `lady` can only be used to call methods in `Dog`
- ▶ `lady.someCockerSpanielMethod()` won't compile

-
- ▶ the **actual type** of the instance determines what definition is used when the method is called

```
Dog lady = new CockerSpaniel();
```

- ▶ `lady.toString()` uses the `CockerSpaniel` definition of `toString`