Inheritance (pt 2)

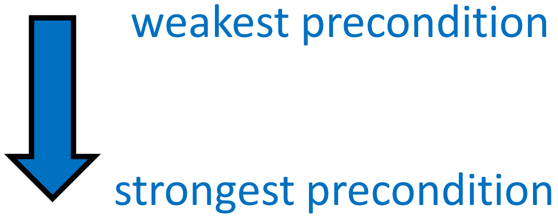
Based on slides by Prof. Burton Ma

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



weakest precondition

strongest precondition

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

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 **`Dog`**s 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

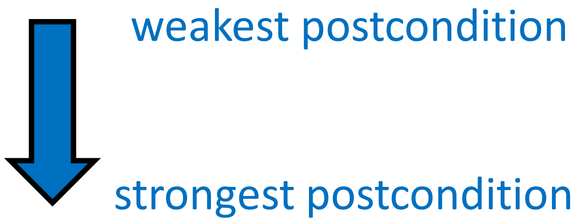
- Postcondition
  - What the method promises to be true when it returns
    - The method might promise something about its return value
      - “Returns size where size is between 1 and 10 inclusive"
    - The method might promise something about the state of the object used to call the method
      - “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()
{ ... }
```



weakest postcondition

strongest postcondition

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

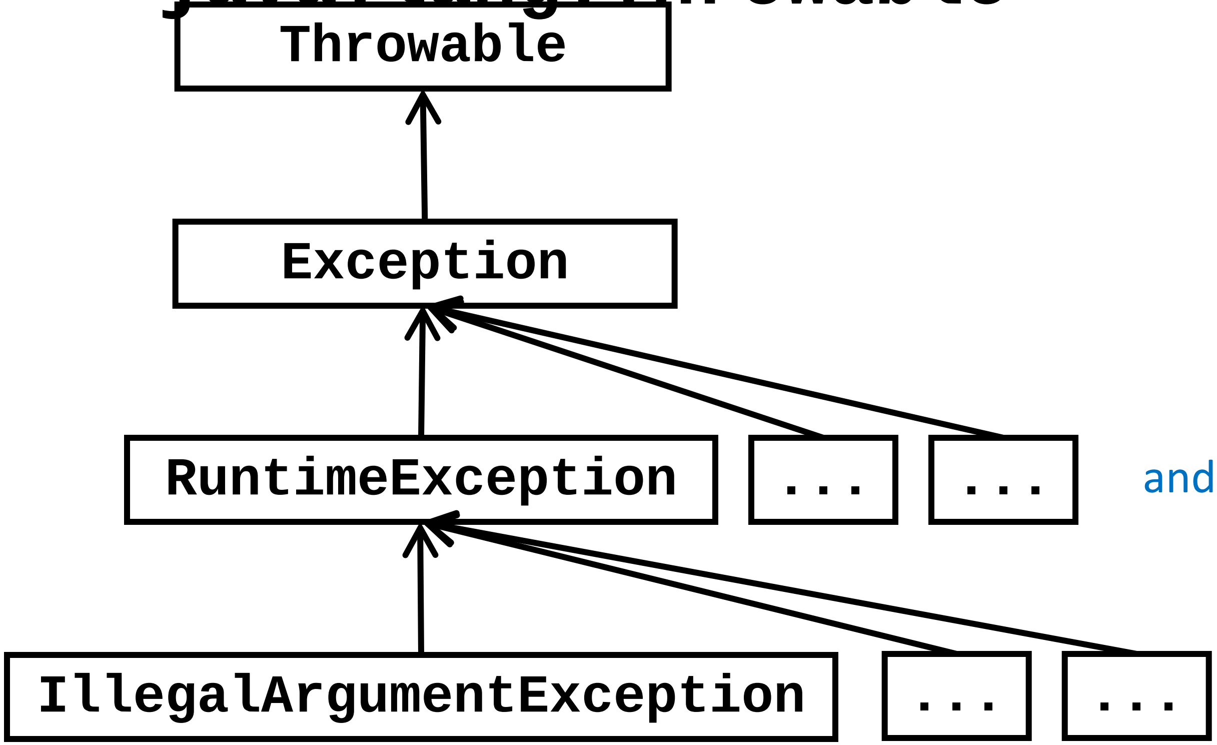
- Client code written for **Dog**s can now fail when given a **Dogzilla**

```
// 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`**



Throwable

Exception

RuntimeException ... ...

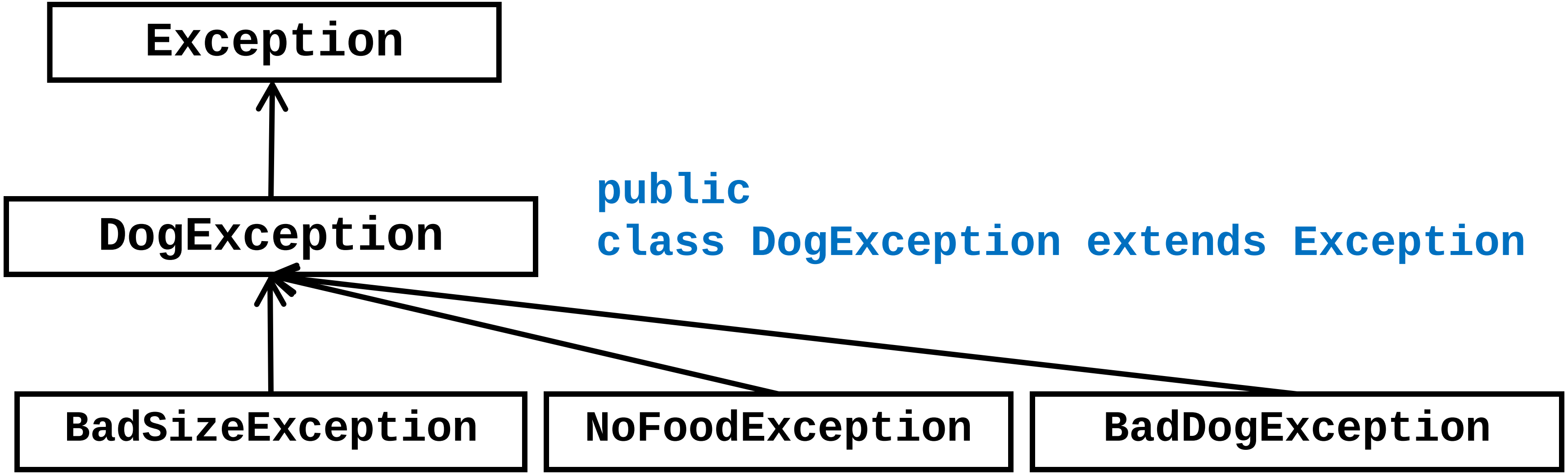
and many, many more

IllegalArgumentException ... ...

and many more

User Defined Exceptions

- You can define your own exception hierarchy
  - Often, you will subclass Exception



Exception

DogException

BadSizeException

NoFoodException

BadDogException

```
public
class DogException extends Exception
```

Exceptions and Inheritance

- A method that claims to throw an exception of type **X** is allowed to throw any 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 an exception of type **X** must also throw an exception of type **X** or a subclass of **X**
  - Remember: a subclass promises to do everything its superclass does; if the superclass method claims to throw an exception then the subclass must also

```
// 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
```

✓

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

Polymorphism

- Inheritance allows you to define a base class that has attributes and methods
  - Classes derived from the base class can use the public and protected base class attributes 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() );
}

// later on...
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());
```

```
Dog toString
CockerSpaniel toString
Mix toString
Date toString
```

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

```
obj.toString()
```

run-time type of the instance **obj**

non-static method

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`**

Abstract Classes

- Sometimes you will find that you want the API for a base class to have a method that the base class cannot define
  - E.g. you might want to know what a **`Dog`**'s bark sounds like but the sound of the bark depends on the breed of the dog
    - You want to add the method bark to **`Dog`** but only the subclasses of **`Dog`** can implement **`bark`**
  - E.g. you might want to know the breed of a **`Dog`** but only the subclasses have information about the breed
    - You want to add the method **`getBreed`** to **`Dog`** but only the subclasses of **`Dog`** can implement **`getBreed`**

Abstract Classes

- Sometimes you will find that you want the API for a base class to have a method that the base class cannot define
  - E.g. you might want to know the breed of a **`Dog`** but only the subclasses have information about the breed
    - You want to add the method **`getBreed`** to **`Dog`** but only the subclasses of **`Dog`** can implement **`getBreed`**

- If the base class has methods that only subclasses can define *and* the base class has attributes common to all subclasses then the base class should be abstract
  - If you have a base class that just has methods that it cannot implement then you probably want an interface
- Abstract :
    - (Dictionary definition) existing only in the mind
- In Java an abstract class is a class that you cannot make instances of

- An abstract class provides a partial definition of a class
  - The subclasses complete the definition

- An abstract class can define attributes and methods
  - Subclasses inherit these
- An abstract class can define constructors
  - Subclasses can call these
- An abstract class can declare abstract methods
  - Subclasses must define these (unless the subclass is also abstract)

Abstract Methods

- An abstract base class can declare, but not define, zero or more abstract methods

```
public abstract class Dog
{
  // attributes, ctors, regular methods

  public abstract String getBreed();
}
```

- The base class is saying "all **`Dog`**s can provide a **`String`** describing the breed, but only the subclasses know enough to implement the method"

Abstract Methods

- The non-abstract subclasses must provide definitions for all abstract methods
  - Consider **`getBreed`** in **`Mix`**

```
public class Mix extends Dog
{ // stuff from before...

  @Override public String getBreed() {
    if(this.breeds.isEmpty()) {
      return "mix of unknown breeds";
    }
    StringBuffer b = new StringBuffer();
    b.append("mix of");
    for(String breed : this.breeds) {
      b.append(" " + breed);
    }
  return b.toString();
}
```

PureBreed

- A purebreed dog is a dog with a single breed
  - One **`String`** attribute to store the breed
- Note that the breed is determined by the subclasses
  - The class **`PureBreed`** cannot give the **`breed`** attribute a value
  - But it can implement the method **`getBreed`**
- The class **`PureBreed`** defines an attribute common to all subclasses and it needs the subclass to inform it of the actual breed
  - **`PureBreed`** is also an abstract class

```
public abstract class PureBreed extends Dog
{
 private String breed;

 public PureBreed(String breed) {
  super();
  this.breed = breed;
 }

 public PureBreed(String breed, int size, int energy) {
  super(size, energy);
  this.breed = breed;
 }
 @Override public String getBreed()
 {
   return this.breed;
 }

}
```

Subclasses of PureBreed

- The subclasses of **`PureBreed`** are responsible for setting the breed
  - Consider **`Komondor`**

Komondor

```
public class Komondor extends PureBreed
{
  private final String BREED = "komondor";

  public Komondor() {
    super(BREED);
  }

  public Komondor(int size, int energy) {
    super(BREED, size, energy);
  }

  // other Komondor methods...
}
```

Static Attributes and Inheritance

- Static attributes behave the same as non-static attributes in inheritance
  - Public and protected static attributes are inherited by subclasses, and subclasses can access them directly by name
  - Private static attributes are not inherited and cannot be accessed directly by name
    - But they can be accessed/modified using public and protected methods

Static Attributes and Inheritance

- The important thing to remember about static attributes and inheritance
  - There is only one copy of the static attribute shared among the declaring class and all subclasses

- Consider trying to count the number of **Dog** objects created by using a static counter

```
// the wrong way to count the number of Dogs created
public abstract class Dog {
  // other attributes...
  static protected int numCreated = 0;

  Dog() {
    // ...
    Dog.numCreated++;
  }

  public static int getNumberCreated() {
    return Dog.numCreated;
  }

  // other contructors, methods...
}
```

protected, not private, so that subclasses can modify it directly

```
// the wrong way to count the number of Dogs created
public class Mix extends Dog
{
  // attributes...

  Mix()
  {
    super();
    Mix.numCreated++;
  }

  // other contructors, methods...
}
```

```
// too many dogs!

public class TooManyDogs
{
  public static void main(String[] args)
  {
    Mix mutt = new Mix();
    System.out.println( Mix.getNumberCreated() );
  }
}
```

prints 2

What Went Wrong?

- There is only one copy of the static attribute shared among the declaring class and all subclasses
  - **`Dog`** declared the static attribute
  - **`Dog`** increments the counter everytime its constructor is called
  - **`Mix`** inherits and shares the single copy of the attribute
  - **`Mix`** constructor correctly calls the superclass constructor
    - Which causes **`numCreated`** to be incremented by **`Dog`**
  - **`Mix`** constructor then incorrectly increments the counter

Counting Dogs and Mixes

- Suppose you want to count the number of **`Dog`** instances and the number of **`Mix`** instances
  - **`Mix`** must also declare a static attribute to hold the count
    - Somewhat confusingly, **`Mix`** can give the counter the same name as the counter declared by **`Dog`**

```
public class Mix extends Dog
{
  // other attributes...
  private static int numCreated = 0;  // bad style

  public Mix()
  {
    super();      // will increment Dog.numCreated
    // other Mix stuff...
    numCreated++; // will increment Mix.numCreated
  }

  // ...
```

Hiding Attributes

- Note that the **`Mix`** attribute **`numCreated`** has the same name as an attribute declared in a superclass
  - Whenever **`numCreated`** is used in **`Mix`**, it is the **`Mix`** version of the attribute that is used

- If a subclass declares an attribute with the same name as a superclass attribute, we say that the subclass attribute hides the superclass attribute
  - Considered bad style because it can make code hard to read and understand
    - Should change **`numCreated`** to **`numMixCreated`** in **`Mix`**