Introduction to Object-Oriented Programming

Review 2: Object-Oriented Programming

Christopher Simpkins
chris.simpkins@gatech.edu

Topics in the OOP Block

- Inheritance
- Polymorphism
- Abstract classes
- Interfaces
- The `equals(Object)` method
- Overriding versus Overloading
- Enums
- Exceptions

Inheritance and Polymorphism

Consider

```
public abstract class Animal { public abstract void speak(); }
public class Mammal extends Animal {
    public void speak() { System.out.println("Hello!"); }
}
public class Dog extends Mammal {
    public void speak() { System.out.println("Woof, woof!"); }
    public void wagTail() { System.out.println("(wags tail)"); }
}
public class Cat extends Mammal {
    public void speak() { System.out.println("Meow!"); }
}
```

We'll use these classes in the examples in the remaining slides.

Assignments

A reference variable has a compile-time type, and refers to an object which has a run-time type.

- The type of the *l-value* (to the left of the = symbol) in an assignment statement is the compile-time type
- The type of the *r-value* (to the right of the = symbol) in an assignment statement is the run-time type
- For reference assignments the type of the *r-value* must be a subclass of the type of the *l-value*
- Remember that every class is a subclass of itself.

So this is fine:

```
Animal fido = new Dog();
```

But this is not:

```
Dog spot = new Mammal();  // Error: Mammal not a subclass of Dog
```

Casting and Method Binding

Casting affects compile-time types (some would say “casting shuts the compiler up”) but method binding is always based on run-time types. So

```
Dog fido = new Dog();
((Mammal) fido).speak();
```

Produces

```
Woof, woof!
```

Even though `Mammal`s say “Hello!” because the run-time type of `spot` is still `Dog`.

Upcasting and Downcasting

The assignment statements we've seen so far are examples of implicit upcasting.

- Upcasting means treating a reference as an instance of one of its superclasses.
- Upcasting is safe becuase every object contains the elements of each of its superclasses.
- Downcasting means treating a reference as an instance of one of its subclasses
- Downcasting is not safe in general because subclasses may add methods not present in superclasses. This is why Java doesn't implicitly downcast in assignment statements.

Think of upcasting as "going up" the class hierarchy and downcasting as "going down" the class hierarchy.

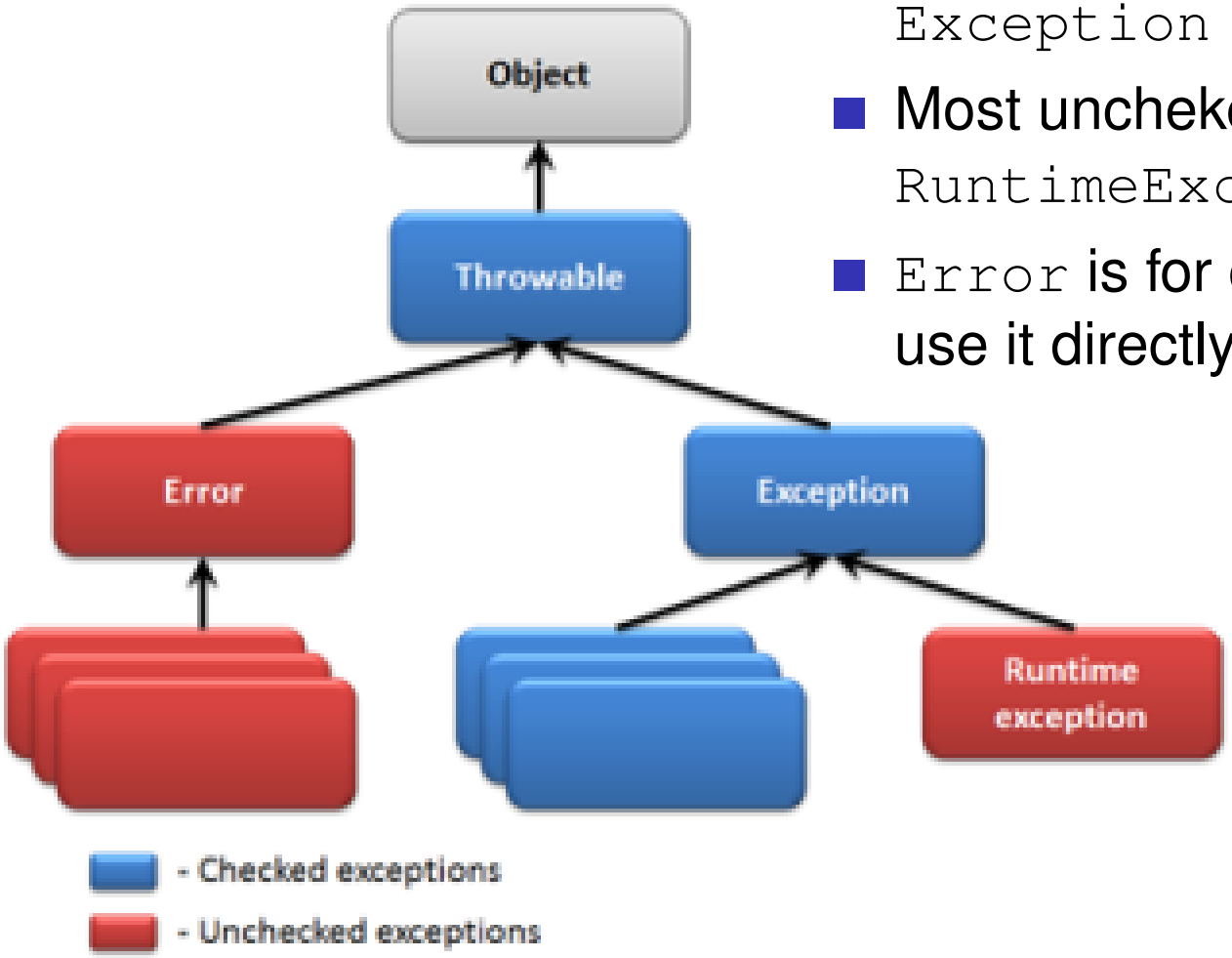
Upcasting and Downcasting Examples

Consider the following:

```
1: Mammal mittens = (Mammal) new Cat(); // Safe
2: Mammal sparky = new Mammal();
3: // Compiles, but will cause a ClassCastException at run-time,
4: Dog huh = (Dog) sparky;
5: // so we won't even get here.
6: huh.wagTail();
```

- The upcast in line 1 is fine.
- The downcast in line 4 will compile but will cause a `ClassCastException` at run-time.
- We won't even get to line 6 due to the exception, which is good because a mammal doesn't have a `wagTail` method. This is what the `ClassCastException` is guarding against.

Java's Exception Hierarchy



- Most (checked) exceptions will subclass `Exception`
- Most uncheked exceptions will subclass `RuntimeException`
- `Error` is for compiler hackers. Don't use it directly.

Catch or Declare

Checked exceptions, sublcasses of `Throwable` that are not subclasses of `RuntimeException`, must be caught or propagated:

Catch:

```
public Company(String employeeDataFile) {
  // ...
  try {
    employees = initFromFile(new File(employeeDataFile));
  } catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
  }
}
```

Declare (propagating the exception):

```
public Company(String employeeDataFile) throws FileNotFoundException {
  // ...
  initFromFile(new File(employeeDataFile));
}
```

Propagating an exception unwinds the stack of methods that led to the point where the exception was thrown.

Exceptions Question

```
public class A extends Throwable { ... }
public class B extends A { ... }
public class C extends RuntimeException { ... }
```

Which of the following methods will **not** compile?

1
```
A foo(B b) throws C {
    if (true) throw new C("c");
    return new B("b");
}
```

2
```
A bar(B b) throws C {
    if (true) throw new RuntimeException("c");
    return new B("c");
}
```

3
```
A baz(B b) throws B {
    if (true) throw new A("a");
    return new B("c");
}
```

Exceptions Question - Answer

```
public class A extends Throwable { ... }
public class B extends A { ... }
public class C extends RuntimeException { ... }
```

Which of the following methods will **not** compile?

1.
```
A foo(B b) throws C {
    if (true) throw new C("c");
    return new B("b");
}
```

2.
```
A bar(B b) throws C {
    if (true) throw new RuntimeException("c");
    return new B("c");
}
```

3. This won't compile because `A` is not a subclass of `B`.

```
A baz(B b) throws B {
    if (true) throw new A("a");
    return new B("c");
}
```

Override Equivalence

Two methods are override-equivalent if:

- they have the same name,
- they have the same parameter lists, and
- their return values are covariant

Override Equivalence Question

Given the following classes:

```
public class A { ... }
public class B extends A { ... }
public class C extends A { ... }
```

and the method signature:

```
public A foo(B b);
```

Which of the following method signatures is override eqivalent?

1. `public B foo(A bar)`
2. `public C foo(B bar)`
3. `public C foo(B bar)`
4. `public B foo(C bar)`

Override Equivalence Question - Answer

Given the following classes:

```
public class A { ... }
public class B extends A { ... }
public class C extends A { ... }
```

and the method signature:

```
public A foo(B b);
```

Which of the following method signatures is override eqivalent?

1. `public B foo(A bar)`
2. **public C foo(B bar)**
3. `public C foo(B bar)`
4. `public B foo(C bar)`