Software "lifecycle" (simplified)

1. Problem statement → requirements analysis
2. Domain analysis
3. System Design
4. Programming (implementing the design)
   - Includes fixing syntax and runtime errors
5. Testing and debugging (not the same thing!)
   - Typically iterate – repeat steps 1 to 5 as necessary
6. Maintenance (could be longest, costliest stage)

Testing

- Means *looking for* bugs
  - Dijkstra: "testing verifies the *presence* of errors, not their absence"
    - i.e., cannot test all possible situations to insure that no bugs remain – but job is to try
- 2 general categories:
  - "Black box testing" – best if by *independent* tester: he/she doesn't know internal structure
  - "White box testing" – *can* be more thorough

Unit testing

- First step of "white box testing"
  - Test each unit separately, before mixing them
    - Each method of each class, each class in each package, each package in each system …
    - Includes testing the `main` method as a unit
  - Test methods by "driver programs"
  - Use "stubs" for incomplete methods
- Next step is integration testing
  - But with confidence that each unit is correct!

Test cases

- Goal: test all possible situations
  - Usually not realistic (unless program is very simple)
- So settle for good test cases
  - Fully test normal functionality – routine cases
    - Be sure to test all branches, even rare ones
  - Include boundary cases (e.g., 0, maximums, …)
  - And remember to test some invalid cases (e.g., not number, negative, …)
    - Good programs should handle gracefully – i.e., don't "crash"

Testing notes

- Coverage testing – an ideal that makes sense: test *each line of code* with at least one test case
- Regression testing – a reality: must *re-run all tests* after every program change
  - Otherwise, likely that bugs are reinserted
  - Need automated tests (e.g., files) to do cheaply
- Other testing: hardware, on-site installation, …
- Tragic truth: testing takes time!
  - But can save time by catching bugs early

Implementing tests

- Some tests can be automatically generated
  - Either systematic intervals, or random inputs
- Much better to use data files – can repeat many tests without much effort
- Sometimes can automatically verify test outputs
  - Maybe find a natural calculation
  - Or maybe find an "oracle" to use
- Or use a testing framework like JUnit

Programming with assertions

- Some testing can be built right in
  - Easy to test assertions – statements that *must be true*
- Java syntax (since SDK 1.4): `assert` *boolean-expression*;
  - If boolean expression is true – assert does nothing
  - But if false – prints a stack trace and exits
- e.g., pre-condition for division – divisor is not 0

```
assert divisor != 0;
return x / divisor; // know it's safe now
```

- Also good for post-conditions, invariants, …

Inheritance

- Can create new classes by extending others
  - New class is called subclass or "child"
  - Extended class is called superclass or "parent"
  - Subclass inherits all of superclass's members
    - And usually has added, or altered features
    - But cannot directly access `private` members
- Results in "is a" relationship
  - Say `class Basketball extends Ball`
    - Then any instance of a Basketball *is a* Ball
    - Reverse is not always true: a Ball can be a Football, or …

Inheritance example from text

`class SavingsAccount extends BankAccount`

- *Inherits* `withdraw`, `deposit`, `getBalance` and `transfer` methods from `BankAccount`
  - Also *has* the instance variable, `balance`, but can't access it directly – it is `private` to `BankAccount`
- Adds `interestRate` variable, `addInterest` method

```
SavingsAccount fund = new SavingsAccount(5);
fund.deposit(1000); // okay – SavingsAccount inherits deposit
momsAccount.transfer(fund, 500); /* okay – the transfer
    method expects a BankAccount type; fund is a BankAccount */
BankAccount general = fund; // okay – a 2nd reference
general.addInterest(); // error – not a BankAccount method
/* even though: */ general instanceof SavingsAccount is true
```

Note: 4 ways to refer to objects

- First 2 ways are trivial:
  - A superclass reference to a superclass object
  - A subclass reference to a subclass object
- 3rd way is safe, but limiting:
  - A superclass reference to a subclass object -

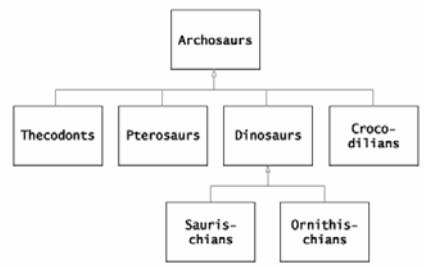
```
BankAccount genfund = new SavingsAccount(5);
```

Now `genfund` can *only* access `BankAccount` methods

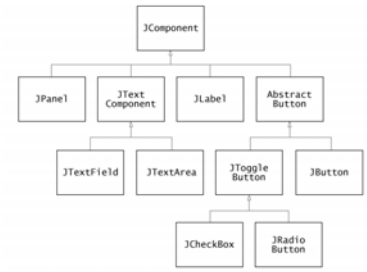
- 4th way is illegal without explicit cast
  - A subclass reference to a superclass object -

```
SavingsAccount mySavings;
mySavings = genfund; // error
mySavings = (SavingsAccount)genfund; // okay
```

Inheritance begets hierarchies



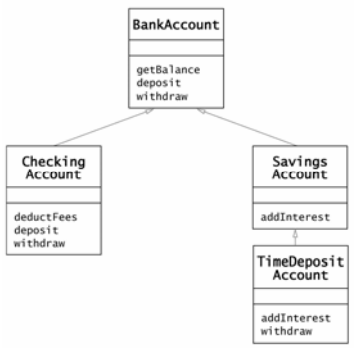
Part of javax.swing hierarchy



Class hierarchies in Java

- Always plain in Java, because each class can only extend *one* other class
  - No platypus-type classes allowed (like in c++)
    - Can implement more than one interface though
  - But subclasses do inherit from superclass parents
    - e.g., if OutdoorBasketball extends Basketball, then an OutdoorBasketball *is a* Basketball *and a* Ball
  - *All* Java classes: descendants of class Object
    - So every object *is an* Object by definition!
- Good hierarchies simplify programming
  - Take advantage of tested code; don't reinvent wheels

A simple bank account hierarchy

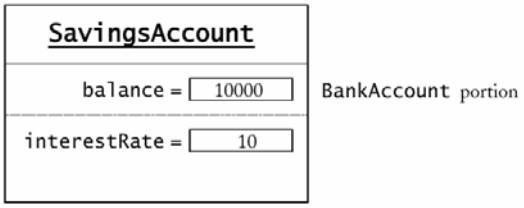


Writing subclasses

- 3 possibilities for instance methods:
  - Inherit – i.e., do nothing
  - Override – have new method act differently
    - Note: use `super` reference to access superclass method
  - Define new – abilities not in superclass at all
- e.g., CheckingAccount (p. 458)
- 2 possibilities for instance variables:
  - Inherit – though if private, must use public methods to access and set
  - Define new – data in addition to superclass data
    - "Shadow variables" – result from trying to override: really just a new variable with the same name – usually a mistake

Constructing a subclass object

- Remember: a *subclass* definition, by itself, just defines *part* of the resulting object



Subclass constructors

- Superclass constructor is *always invoked first*
  - i.e., call to `super` is always the first statement of a subclass constructor
  - If not done explicitly, it will happen *implicitly*
    - The compiler puts it there if you don't!

    `super();` // so superclass *must* have no-arg constructor
  - Explicit call necessary to use a different superclass constructor – e.g., see CheckingAccount.java again
- FYI: superclass `finalize()` is *always last* too

Writing classes to be extended

- Always provide a no-argument constructor
- Control subclass access as appropriate
  - Already know about `private` and `public`
  - `protected` – *only* subclasses and other classes in the same package can access
  - (package) – *only* classes in same package can access
    - A.k.a. "friendly" or default access (often omitted by mistake)
- Also can inhibit subclass abilities with `final`
  - `final class` – cannot be extended (e.g., `String`)
  - `final` method – subclasses cannot override